Chapter 16: Security

With the phenomenal level of interconnectedness prevalent in the modern world, opportunities arise not just in legitimate commercial enterprises, but also in the more clandestine establishments — individuals or organisations that seek to capitalise on unprotected systems and networks, with an overarching motive to threaten their targets for the purpose of extortion, cause immediate and lasting harm, or directly profit from illicit activities. The topic of conversation is, of course, cybercrime and information security.

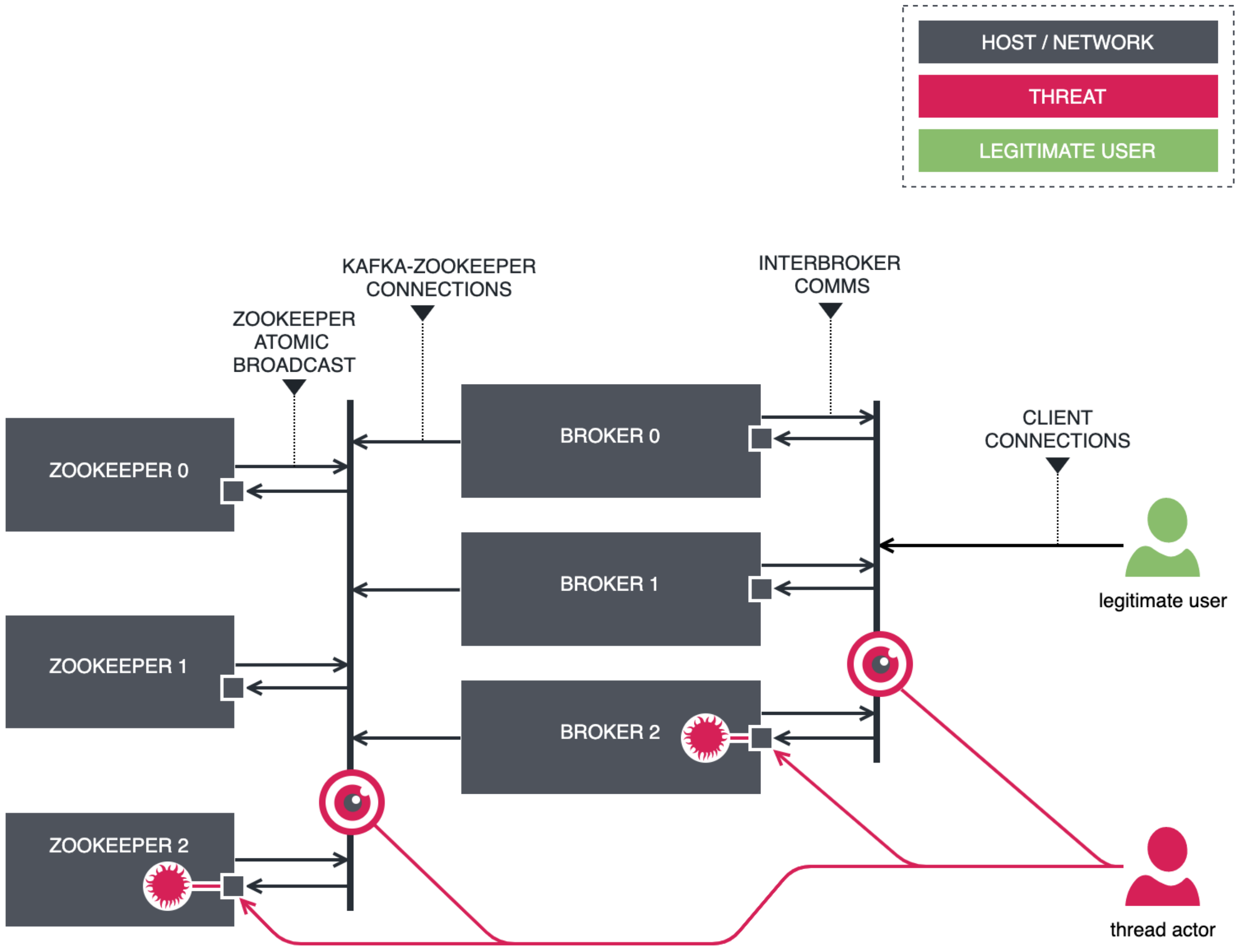
There is a common misconception that cybercriminals target an organisation's most secure defences and employ highly sophisticated techniques to penetrate defences. This view has been largely popularised in modern fiction, where the persona of a hacker has been romanticised as a lone, reclusive individual with above-average skills who acts out of spite or takes their target as a personal challenge. In practice, it is more instructive to view cybercrime as a business, albeit an illegitimate one. While just about any target is theoretically 'hackable', some are just not worth it, insofar as the cost of breaking into an organisation may outweigh the prospective gains. Like any business, a criminal organisation will seek to profit with minimal outlay, which means going after the easy prospects, with trivial or non-existent defences. And where an organisation is heavily defended, cybercriminals will often probe for alternate ways in, looking for weaknesses at the perimeter, as well as from inside the organisation — using social hacking to gain access. As such, it is imperative to utilise multi-layered defences, creating redundancy in the event a security control fails or a vulnerability is exposed

Kafka is a persistent data store that potentially contains sensitive organisational data. It often acts as a communication medium, connecting disparate systems and ensuring seamless information flow within the organisation. These characteristics make Kafka a lucrative target to cybercriminals: if breached, the entire organisation could be brought to its knees. As such, it is essential that particular attention is paid to the security of Kafka deployments, particularly if the cluster is being hosted among other infrastructure components or if it is accessible from outside the perimeter network.

State of security in Kafka

Let us start by making one thing abundantly clear: *Kafka is not secure by default.* Kafka has numerous security controls that cover virtually all aspects of information security; however, these controls are disabled out of the box.

The default Kafka security profile is illustrated below.



Out-of-the-box state of Kafka security

Specifically, the notable areas of shortfall are:

- **Any client can establish a connection to a ZooKeeper or Kafka node.** This statement doesn't just cover the conventional ports `2181` (ZooKeeper) and `9092` (Kafka) — diagnostic ports such as those used by JMX (Java Management Extensions) are also available for anyone to connect to. What makes this setup particularly troubling is that a connection to ZooKeeper is not essential for the correct operation of legitimate clients. While earlier versions of Kafka required clients to interface directly with ZooKeeper, this requirement has been removed as of version 0.9 — a release that dates back to November 2015. ZooKeeper is a highly prized trophy for potential attackers, as writing to *znodes* (internal data structure used to persist state within ZooKeeper) will trivially compromise the entire cluster.
- **Connections to Kafka brokers are unencrypted.** Connections are established over TCP, with the parties exchanging data over a well-documented binary protocol. A third party with access to the transit network may use a basic packet sniffer to capture and analyse the network traffic, obtaining unauthorized access to information. A malicious actor may alter the information in

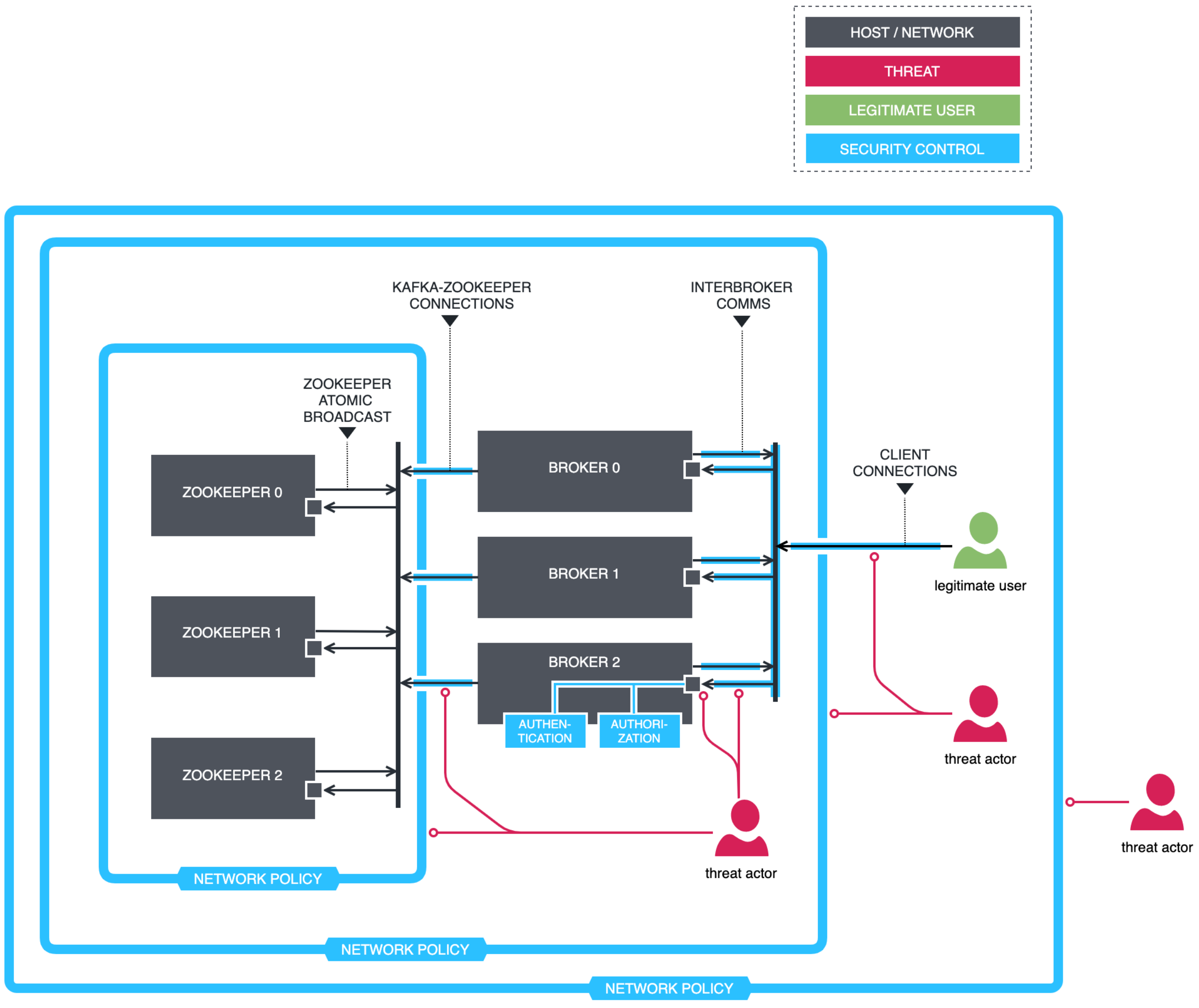
transit or impersonate either the broker or the client. In addition to client traffic, interbroker traffic is also insecure by default.
- **Connections to Kafka brokers are unauthenticated**. Any client with knowledge of the bootstrap endpoints can establish a connection to the brokers, appearing as a legitimate client to Kafka. A client does not have to identify itself, nor prove that the supplied identity is genuine.
- **No authorization controls are in place**. Even if authentication is enabled, a client may perform any action on a broker. The default authorization policy is *allow-all*, letting a rogue client run amok.

The reason for Kafka's relaxed default security profile is largely historical. Early versions of Kafka had no support for security, and introducing mandatory security controls would have impacted compatibility with existing clients.

Target state security

Before embarking on the journey of progressively hardening the cluster, it is worthwhile to model an ideal target state. This will act as a notional reference, allowing us to chart a course and ensure that we stick to it.

An overview of the target state is illustrated below. A further examination of the various security principles follows.



Target state security

Minimise the attack surface

There are several ways clients can connect to the Kafka and ZooKeeper cluster, and most of these ingress points are of no use to legitimate clients. On the flip side, they create a large attack surface area, making it difficult to enforce. As such, we need to limit access to the cluster at the lowest possible level, exposing the minimally-essential set of endpoints for legitimate clients.

In addition to limiting ingress, we may also want to explicitly allow specific groups of clients. We might only allow clients from the organisation's internal network to access the cluster. But we may go further than that, segregating traffic based on the clients' location within the internal network. For example, we may partition the internal network into multiple segments, and only allow clients deployed in the operational network (such as the data centre) to access Kafka, while blocking clients in the general-purpose (office) network.

There may also be legitimate reasons to allow external clients to connect; for example, if those clients reside on a remote network that is also trusted, but the two networks are separated by an untrusted network, such as the public Internet. Often this scenario occurs in a hybrid deployment topology, where some parts of an organisation's IT systems may be deployed in one or more data centres, while other parts may be deployed in the public Cloud, or multiple Clouds in different geographical regions, for that matter.

Ensure traffic confidentiality

Traffic flowing in and out of the cluster, and within the cluster, needs to be protected from eavesdropping and tampering. Ordinarily, this implies some sort of encryption, a message signing protocol, and a secure key exchange protocol that collectively assure the end-to-end confidentiality of messages.

Encryption protocols must be chosen such that they comply with industry-accepted standards for the secure exchange of information. These standards exist for a broad set of applications, curated by organisations such as the National Institute of Standards and Technology (NIST). Specifically, NIST breaks standards down into cryptographic hash functions, symmetric key algorithms, and asymmetric key algorithms — all of which will apply to Kafka at some point during the connection lifecycle. In addition to general-purpose standards, specific industry bodies — such as PCI, as well as legislative acts — such as HIPPA, GLBA and SOX, will mandate security controls in addition to the baseline. When deploying Kafka for applications that are expected to comply with regulation, care must be taken to ensure that the applications' approach to information security is equally compliant.

Know the client

Kafka clients are typically embedded into or act on behalf of other applications that play a role in the overall architecture landscape. Identifying clients is a prerequisite for access control; if we don't know who our clients are, we cannot possibly guarantee that access has been restricted to the minimum set of actions that a client is reasonably expected to undertake. A reader familiar with the principles of information security will recognise this statement as the *Principle of Least Privilege* (PoLP). To be specific, attesting the identity of a client does not in itself realise PoLP, but is nonetheless essential for subsequent controls whose role it is to categorically enforce this principle.

Limit access to essential data and functionality

This is the direct enforcement of PoLP. We need to identify all prospective principals — users of the cluster — and determine which resources in the cluster they are allowed to operate upon. By 'resources', it is meant any entity that may pose a material threat if consumed uncontrollably. These include topics, consumer groups, brokers, configuration entries, as well as users. The latter is a resource like any other — the uncontrolled modification of users' profiles may lead to a *privilege escalation*, where a less privileged user may gain a level of access above what has officially been allocated to them. It may also lead to a *denial of service*, where a legitimate user has been blocked from carrying out their normal functions.

With an aspirational target state defined, the scene is set — the rest of the chapter is focused on fulfilling these aspirations — getting Kafka to a state of acceptable security.

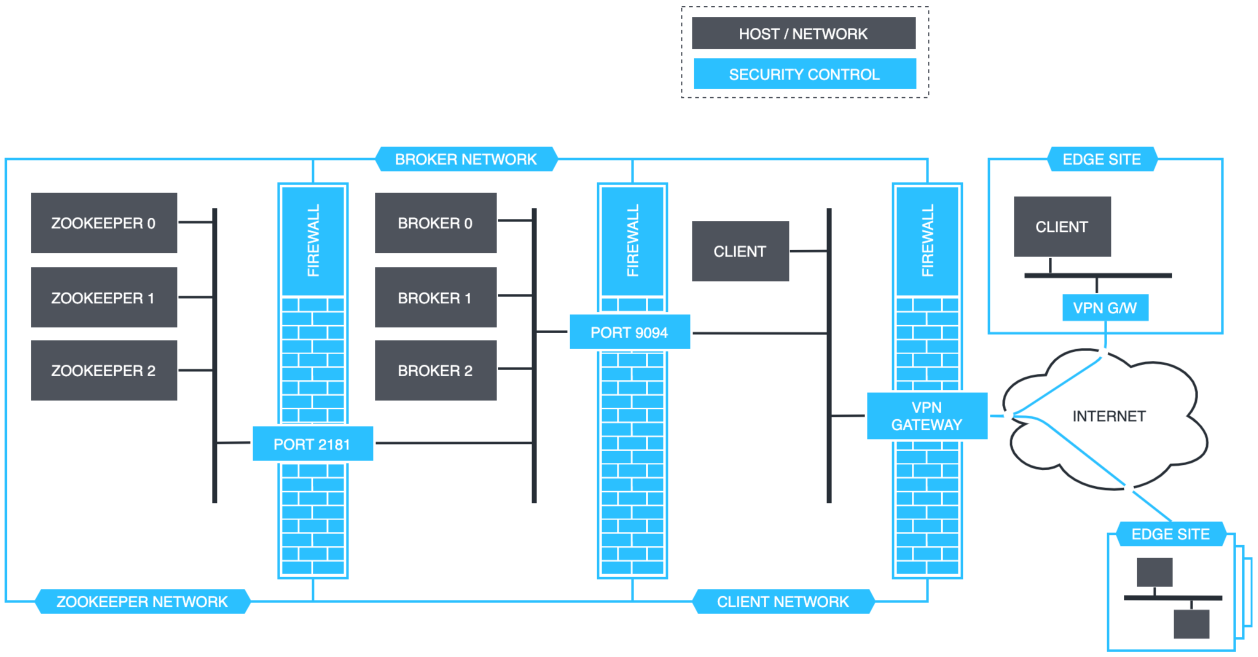
Network traffic policy

Speaking to the earlier point on reducing the attack surface, no control is more effective than a network-level traffic policy — or as it is more commonly known, a firewall.

> Without burrowing into the details, the defining role of a firewall is to segregate traffic between two network segments, such that only legitimate traffic is permitted to traverse the firewall. Conversely, illegitimate traffic (which may be malicious or accidental) is not permitted to pass through the firewall. Network and transport-level controls tend to be simple, straightforward rules that restrict the flow of packets. Because of their simplicity, they are both efficient and difficult to bypass. Especially when used in conjunction with a default-deny policy, one can confidently reason about the behaviour of a firewall, and more importantly, verify its behaviour using real and simulated traffic.

This chapter is not going to examine the different firewall types and how one would practically configure and maintain a firewall. The technology options and physical deployment topologies are just too varied. The physical architecture will further be complicated with the introduction of geographical regions or the use of multiple redundant hosting locations. Rather than attempting to prescribe a physical topology, the intention is to capture the network architecture at a logical level.

At a minimum, the following network partitioning scheme is recommended.



Network segregation

The reason that the firewall port on the broker network is listed as 9094 rather than 9092 is to accommodate an alternate listener, blocking the default one. More on that later.

In the arrangement above, the network is partitioned into four static segments. The ZooKeeper ensemble operates within its own, heavily isolated network segment. This might be a subnet or a series of mutually-routable subnets. The latter would be used in a multi-site deployment, where disparate physical networks are used to collectively host the ensemble. One may also use an overlay network, or other forms of virtual networks, to merge physically separate networks into a single logical network. Again, we must abstain from the detail at this point, and stick to the logical viewpoint.

The brokers operate within a dedicated network segment, which is separated from the ZooKeeper segment by way of a stateful firewall. This ensures that legitimate connections can be established from the brokers to the ZooKeeper nodes, but only those connections and nothing else. No other network segment is permitted to access the ZooKeeper segment. In practice, there may be a need to make specific exclusions for administrative purposes, so that authorized operators of Kafka are able to make changes to the underlying ZooKeeper configuration, deploy software updates, and so forth.

The next outermost segment is the internal client network. This includes producer, consumer and admin clients — essentially the full suite of internal applications that rely on Kafka to shuttle event data. There may be multiple such client networks, or the client network may be fragmented internally. This often happens in hybrid deployment topologies involving different sites. But the topology is more or less stable; in other words, we don't expect entire sites to come and go on a whim.

The fourth network segment is everything outside the perimeter, which can simply be treated as the external network. In the simplest case, the external network is the public Internet. However, this is a relative term. In more elaborate segregation topologies, the external segment might still be within the organisation's perimeter, but outside the client network. It may be the general-purpose staff network or some other corporate intranet. And while in a relative sense, this network might be more trusted than the public Internet, its level of assurance is still largely classified as a 'walk-in' network — it should not be allowed to interface directly with the key technology infrastructure underpinning the revenue-generating systems that propel the organisation.

In addition to the static segments, there may be a case for one or more dynamic network segments for the client ecosystem that could appear and disappear at short notice. A common use case is *edge computing*, where sites may be spun up 'closer to the action', so to speak, but the sites themselves might not be long-lived or may change their location. There may be other ways to allow remotely-deployed applications to utilise Kafka — for example, via dedicated APIs. However, Kafka does not preclude remote access and in some cases this may be desirable — for example, the remote site may have a requirement to persist events internally but might not have a local deployment of Kafka at its convenience. Edge computing dovetails nicely into Kafka's authorization controls, which will be discussed later.

Perhaps the most common use case for remote connectivity is telecommuting. Engineers may be working from a variety of locations, which could be a mixture of private networks, public network, fixed location and mobile networks. Clearly, these sites cannot be easily controlled and are therefore labelled as untrusted. Nonetheless, telecommuters will require first-class access to backend systems and key technology infrastructure to maintain productivity. Although some remote sites may be untrusted, the individual hosts may still be trusted — for example, a remote worker accessing the corporate network over a free Wi-Fi in a café.

Edge locations that require direct connectivity into the core client network are best accommodated using secure virtual networks, such as VPNs. Rather than maintaining temporary 'pinhole' firewall rules that allow access from specific locations based on origin network addresses (which may not always be discernible, particularly if the site is behind a NAT device), a VPN can be used to securely span physically separate networks. The sites (or individual hosts) would be dispensed individual credentials, which may be distributed in the form of client-side Digital Certificates. These credentials will be used to authenticate the site (or host) to the central VPN gateway. The VPN gateway typically resides behind the firewall, although some firewall vendors combine VPN and packet-filtering capabilities into a single appliance. (The latter is sometimes called a VPN concentrator.) Kafka need not be aware of edge locations and VPN arrangements. Instead, the responsibility of accommodating remote sites falls upon the client network and its maintainers. As long as remote sites can securely attach to and transit through the client network, they will have access to the Kafka cluster.

Confidentiality

Kafka supports Transport Layer Security (TLS) for encrypting traffic between several key components. Unlike the discussion on network traffic policy, which maintained a purely theoretical stance, this section offers hands-on examples for configuring TLS and securely connecting producer and consumer clients over encrypted links.

> The reader may be familiar with TLS just by virtue of browsing websites. You may have noticed the padlock on the address bar of the browser — this is an indication that the browser is communicating over an encrypted connection. HTTPS is increasingly used in place of HTTP to secure websites; the 'S' in HTTPS stands for 'secure' and implies the use of SSL and TLS algorithms. In nutshell, SSL is obsolete and TLS is the new name of the older SSL protocol. Technically, the term TLS is more accurate, but most people still use SSL. Digital Certificates used to verify the communicating parties are often referred to as 'SSL Certificates', but the name is purely historical — in fact, they can be used to secure both SSL and TLS traffic.

Like most parts of Java and the rest of the world, Kafka uses the term SSL to refer to TLS. The latest version of TLS at the time of writing is TLS v1.3. The latest version supported by Kafka is TLS v1.2. From a security standpoint, the differences between v1.3 and v1.2 are generally considered to be minor — v1.3 removes certain deprecated ciphers that have known exploits, reducing the likelihood of a misconfiguration impacting the security posture. Where v1.3 trumps its predecessor is in the area of performance — specifically, in connection establishment time.

At the time of writing, support for TLS v1.3 is scheduled to be introduced in version 2.5.0 of Kafka. The main reason for the delayed adoption of TLS v1.3 in Kafka is that it introduces a dependency on Java 11, which is a step up from its current reliance on Java 8, but also a break in runtime compatibility. All in all, when correctly configured, TLS v1.2 is considered to be adequate for general use, and mandated by the Payment Card Industry (PCI) Security Standards Council for the Data Security Standard (DSS).

An essential element of TLS is an X.509 Certificate, often referred to as 'Digital Certificate' or simply 'certificate'. At a minimum, a certificate authenticates the server side of a TLS connection to the client, the latter being the entity that initiates the connection, while the former accepts the connection on a TLS socket. In our case, the server is a broker node. By verifying the certificate presented by the broker to the client against either a trusted certificate authority or a pre-agreed self-signed certificate, the client can be assured that the broker is, in fact, who it claims to be. Certificates can also be used in the opposite direction, attesting the identity of a client to the broker. This scenario will be covered later, in the course of authentication.

> In cryptography, X.509 is a standard defining the format of public-key certificates. X.509 certificates are used in TLS/SSL, which is the basis for HTTPS. They are also used in offline applications that

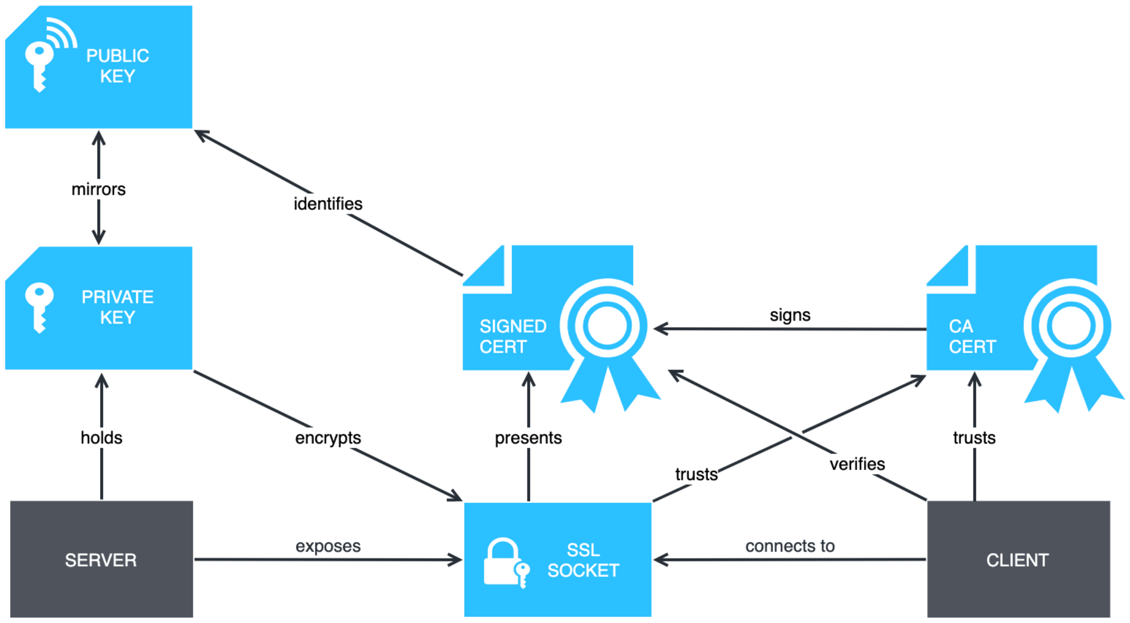
require tamper-proof electronic signatures. An X.509 certificate contains a public key and an identity (a hostname, an organisation, or an individual) — this is depicted in either the Common Name (CN) attribute of the certificate or the Subject Alternative Name (SAN), the latter being an extension to the X.509 base standard.

A certificate is either signed by a *Certificate Authority* (CA) or self-signed. When a certificate is signed by a trusted certificate authority, or validated by other means, someone holding that certificate can rely on the public key it contains to establish secure communications with another party, or validate documents digitally signed by the corresponding private key.

A certificate authority is akin to a government office that issues passports. Passports are printed on specially-crafted materials and use various stamps, watermarks, ultraviolet ink, magnetic strips and holograms to make them difficult to forge. Other governments may inspect the passport using the verification means at their disposal to ascertain its authenticity. Whereas a passport predominantly relies on physical means for ensuring security, the signing of X.509 certificates achieves the equivalent using cryptographic algorithms, with the certificate authority acting as the digital equivalent of a passport office. For as long as the CA remains a genuine and trusted authority, the clients have the assurance that they are connecting to authentic parties. (More recently, passports have also started embedding an RFID chip with a cryptographic module that holds digitally signed biometric data of the passport holder — adding digital security to a traditionally physical device.)

Complementary to X.509 certificates is *Public Key Infrastructure* (PKI) — a comprehensive set of roles, policies, hardware, software and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption. PKI is an arrangement that binds public keys with respective identities of entities (individuals, hosts and organisations). The binding is established through a process of registration and issuance of certificates at and by a CA. Depending on the assurance level of the binding, this may be carried out by an automated process or under human supervision.

The diagram below illustrates the relationship between a pair of communicating parties — a client and a server — and the supporting mechanisms by which the integrity and confidentiality of the information exchange is guaranteed.



Securing communications using SSL/TLS

It should be stated from the outset that the use of a Certificate Authority and PKI is unequivocally preferred over self-signed certificates. It is not that CA-signed certificates are inherently more secure from a purely cryptographic perspective; rather, the use of PKI makes it easier to manage large numbers of certificates, streamlining the process of issuing, rotating, and revoking certificates. This aspect makes CA-signed certificates more robust overall, as the likelihood of an error is greatly reduced, compared to the bespoke process of managing self-signed certificates. In addition, self-signed certificates must be exchanged out of band, before the parties can communicate securely. This increases the likelihood of them being intercepted and modified en route, which also erodes the trust one may place in self-signed certificates.

For the upcoming examples, we are going to generate our own CA for signing certificates. Naturally, we would have proceeded with a complete PKI scenario, but in practice, the choice of PKI technology elements and their deployment options will be independent to the rest of technology infrastructure choices. Rather than distract the reader with the nuances of PKI, the focus will be on getting a secure connection up and running, with the blissful assumption that PKI is someone else's problem.

To work through the examples below, you will need `openssl` and `keytool` installed. These are open-source packages that will either be pre-installed or can easily be downloaded through a package manager. When generating keys and certificates, you will end up with sensitive material that shouldn't be left unattended, even if it is on your local machine. As such, it is best to create a dedicated directory for all intermediate operations, then delete the directory once you are done. In the upcoming examples, we will be working out of `/tmp/kafka-ssl`.

Java applications use *keystore* and *truststore* files to store keying material. The contents of these files are encoded in a format specific to the Java ecosystem. Keystore files hold private keys and the associated signed certificates, presented by the client or server applications that hold them. Truststore files house the trusted certificates to authenticate the opposing party. Both files are password-protected. In the case of a keystore, individual keys may be password-protected. The examples in this chapter use the password `secret`. This should be substituted with a more appropriate string.

All examples use a validity period of 365 days for keys and certificates. This period can easily be changed by specifying an alternate value for the `-days` flag (to the `openssl` command) or the `-validity` flag (to `keytool`).

Client-to-broker encryption

We need to generate a key and certificate for each broker in the cluster. The common name of the broker certificate must match the fully qualified domain name (FQDN) of the broker, as the client compares the CN with the resolved hostname to make sure that it is connecting to the intended broker (instead of a malicious one). This process is called *hostname verification*, and is enabled by default. In our examples, we will assume that we are targetting our local test cluster, and therefore the CN will simply be `localhost`. We could use an arbitrary hostname or one containing a wildcard, provided it matches the hostname used in the bootstrap list, as well as the corresponding hostname declared in the `advertised.listeners` broker property.

The client may disable hostname verification, which will allow it to connect to any host, provided that its authenticity can be verified by traversing the certificate chain. However, disabling hostname verification can constitute a serious security flaw, as it makes the client trust *any* certificate that was issued by the CA. This may not be cataclysmic if using a private CA that is constrained to an organisation's PKI; but when a certificate is issued by a public CA, the absence of hostname verification allows any party to impersonate a broker, provided it can hijack the DNS. *Hostname verification is an integral part of server authentication.* Disabling hostname verification may only be safely done in non-production workloads, and even then this practice should be discouraged as it may inadvertently lead to a misconfiguration of production deployments.

Generate the private key

To begin, we need to generate an SSL key and certificate for each broker. We have just one in our test cluster; otherwise, this operation would have to be repeated. (Or better still — automated.) Run the following:

```
keytool -keystore server.keystore.jks -alias localhost \
    -validity 365 -genkey -keyalg RSA
```

This command will prompt you for a password. After entering and confirming the password, the next prompt is for the first and last name. This is actually the common name. Enter `localhost` for our test cluster. (Alternatively, if you are accessing a cluster by a different hostname, enter that name instead.) Leave other fields blank. At the final prompt, hit `y` to confirm.

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=localhost, OU=Unknown, O=Unknown, L=Unknown, 
    ST=Unknown, C=Unknown correct?
  [no]:  y

Generating 2,048 bit RSA key pair and self-signed certificate 
    (SHA256withRSA) with a validity of 365 days
  for: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, 
      ST=Unknown, C=Unknown
```

The result will be a `server.keystore.jks` file deposited into the current directory.

You can view the contents of the keystore at any time — by running the following command. (It will require a password.)

```
keytool -list -v -keystore server.keystore.jks
```

```
Enter keystore password:
Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: localhost
Creation date: 02 Jan. 2020
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, 
    ST=Unknown, C=Unknown
Issuer: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ▯
    ST=Unknown, C=Unknown
Serial number: 2e0b92ac
Valid from: Thu Jan 02 08:51:53 AEDT 2020 until: ▯
    Fri Jan 01 08:51:53 AEDT 2021
Certificate fingerprints:
   SHA1: 59:2D:AA:C0:A8:B1:CF:B6:F7:CA:B6:C2:21: ▯
      55:44:12:27:44:0F:58
   SHA256: 27:F5:7F:9E:36:A1:B4:0D:72:F6:71:AC: ▯
      A0:8B:F2:BB:06:CA:0C:FD:28:64:86:53:6A:37:BF:EF:81:D0:7F:68
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: B2 00 B4 C1 BA 4A 5E FC   9B 44 B7 29 F3 78 A2 CD
0010: 4A BA 6D 4E
]
]

*******************************************
*******************************************
```

Create a CA

Bearing a private key alone is insufficient, as it does not identify the user of the key or instil trust in it. The following step creates a certificate authority for signing keys.

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

You are required to provide a password, which may differ from the password used in the previous step. Leave all fields empty with the exception of the *Common Name*, which should be set to `localhost`.

```
Generating a 2048 bit RSA private key
.........+++
.............+++
writing new private key to 'ca-key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a
Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:
State or Province Name (full name) []:
Locality Name (eg, city) []:
Organization Name (eg, company) []:
Organizational Unit Name (eg, section) []:
Common Name (eg, fully qualified host name) []:localhost
Email Address []:
```

The above command will output `ca-key` and `ca-cert` files to the current directory.

The next two steps will import the resulting `ca-cert` file to the broker and client truststores. Once imported, the parties will implicitly trust the CA and any certificate signed by the CA.

```
keytool -keystore client.truststore.jks -alias CARoot \
    -import -file ca-cert
```

```
Enter keystore password:
Re-enter new password:
Owner: CN=localhost
Issuer: CN=localhost
Serial number: 8f444e15ad8f7067
Valid from: Thu Jan 02 09:28:45 AEDT 2020 until: ⏎
    Fri Jan 01 09:28:45 AEDT 2021
Certificate fingerprints:
   SHA1: 70:55:42:23:69:A1:EA:E8:13:49:41:CC:C3:CE:⏎
      A3:7B:CB:25:F8:08
   SHA256: 7E:CC:21:57:5B:8C:FB:90:D9:9E:2B:84:76:⏎
```

```
      C4:E1:83:D0:2D:B5:D1:17:3A:D2:D5:5A:4D:C5:CB:F3:9B:32:DD
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Trust this certificate? [no]:  y
Certificate was added to keystore
```

Repeat for `server.truststore.jks`:

```
keytool -keystore server.truststore.jks -alias CARoot \
    -import -file ca-cert
```

Generating private keys and the creation of a CA are mutually independent operations and may be performed in either order. The results of these operations will be combined in the signing stage.

Sign the broker certificate

The next step is to generate the certificate signing request on behalf of the broker.

```
keytool -keystore server.keystore.jks -alias localhost \
    -certreq -file cert-req
```

This produces `cert-req`, being the signing request. To sign with the CA, run the following command.

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-req \
    -out cert-signed -days 365 -CAcreateserial
```

```
Signature ok
subject=/C=Unknown/ST=Unknown/L=Unknown/O=Unknown/ 
    OU=Unknown/CN=localhost
Getting CA Private Key
Enter pass phrase for ca-key:
```

This results in the `cert-signed` file.

The CA certificate must be imported into the server’s keystore under the `CARoot` alias.

```
keytool -keystore server.keystore.jks -alias CARoot \
    -import -file ca-cert
```

```
Enter keystore password:
Owner: CN=localhost
Issuer: CN=localhost
Serial number: 8f444e15ad8f7067
Valid from: Thu Jan 02 09:28:45 AEDT 2020 until: ▯
    Fri Jan 01 09:28:45 AEDT 2021
Certificate fingerprints:
   SHA1: 70:55:42:23:69:A1:EA:E8:13:49:41:CC:C3:CE:A3: ▯
      7B:CB:25:F8:08
   SHA256: 7E:CC:21:57:5B:8C:FB:90:D9:9E:2B:84:76:C4: ▯
      E1:83:D0:2D:B5:D1:17:3A:D2:D5:5A:4D:C5:CB:F3:9B:32:DD
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Trust this certificate? [no]:  y
Certificate was added to keystore
```

Then, import the signed certificate into the server's keystore under the `localhost` alias.

```
keytool -keystore server.keystore.jks -alias localhost \
    -import -file cert-signed
```

Resulting in:

```
Enter keystore password:
Certificate reply was installed in keystore
```

Deploy the private key and signed certificate to the broker

With key generation and signing operations completed, the next step is to install the private key and the signed certificate on the broker. Assuming the keystore file is in `/tmp/kafka-ssl`, run the following:

```
cp /tmp/kafka-ssl/server.*.jks $KAFKA_HOME/config
```

Configure the broker to use SSL

Edit `server.properties` and make changes to reflect the following.

```
listeners=PLAINTEXT://:9092,SSL://:9093
advertised.listeners=PLAINTEXT://localhost:9092,SSL://localhost:9093
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,\
    SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
inter.broker.listener.name=PLAINTEXT
ssl.keystore.location=\
    /Users/me/opt/kafka_2.13-2.4.0/config/server.keystore.jks
ssl.keystore.password=secret
ssl.key.password=secret
ssl.truststore.location=\
    /Users/me/opt/kafka_2.13-2.4.0/config/server.truststore.jks
ssl.truststore.password=secret
```

Looking over the configuration —

- The first change is the addition of `SSL://:9093` to the `listeners` list. This creates a new server socket, bound to port `9093`.
- The socket is advertised as `SSL://localhost:9093` in the `advertised.listeners`.
- The `listener.security.protocol.map` and `inter.broker.listener.name` properties remain unchanged for this example, as we have not defined a new protocol, nor have we changed how the brokers communicate with each other.
- The addition of `ssl...` properties configures the broker to use the keying material that was generated in the previous steps.

Having saved `server.properties`, restart the broker for the changes to take effect.

In the configuration above, the SSL settings were defined in global scope — applying uniformly to all listeners configured to use TLS, including the interbroker connection (if applicable). Kafka allows us to specify custom SSL settings for individual listeners, by prefixing the `ssl...` property names with the lowercase name of the listener, in the form `listener.name.<lowercase_name>.<setting>=<value>`; for example, `listener.name.external.ssl.keystore.password=secret`.

Deploy the CA certificate to the client

The previous client examples were communicating with the broker over a cleartext connection. In order to enable SSL, we must first copy the `client.truststore.jks` file to our source code directory:

```
cp client.truststore.jks ~/code/effectivekafka
```

Configure the client to use SSL

The complete source code listings for the SSL producer and consumer client examples are available at github.com/ekoutanov/effectivekafka[31] in the `src/main/java/effectivekafka/ssl` directory. The keystore and truststore files have not been committed to the repository, as they vary between deployments.

The following example configures a producer client to use SSL. With the exception of new SSL-related configuration properties, it is otherwise identical to the original producer sample presented in Chapter 5: Getting Started.

```
import static java.lang.System.*;

import java.util.*;

import org.apache.kafka.clients.*;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.config.*;
import org.apache.kafka.common.serialization.*;

public final class SslProducerSample {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "getting-started";

    final Map<String, Object> config = Map
        .of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9093",
            CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
            "SSL",
            SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
            "https",
            SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
            "client.truststore.jks",
            SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,
            "secret",
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName(),
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName(),
            ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
```

[31]https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/ssl

```
          true);

    try (var producer = new KafkaProducer<String, String>(config)) {
      while (true) {
        final var key = "myKey";
        final var value = new Date().toString();
        out.format("Publishing record with value %s%n",
                   value);

        final Callback callback = (metadata, exception) -> {
          out.format("Published with metadata: %s, error: %s%n",
                     metadata, exception);
        };

        // publish the record, handling the metadata in the callback
        producer.send(new ProducerRecord<>(topic, key, value),
                      callback);

        // wait a second before publishing another
        Thread.sleep(1000);
      }
    }
  }
}
```

The first point of difference is the use of the `security.protocol` property, which specifies the use of SSL. This property holds a compound value — not only does it specify the encryption scheme, but also the authentication scheme. As we are not using authentication in this example, its value is simply `SSL`.

The `ssl.endpoint.identification.algorithm` property specifies how the broker's hostname should be validated against its certificate. The default value is `https`, enabling hostname verification. (This is the recommended setting.) To disable hostname verification, set this property to an empty string.

The `ssl.truststore.location` and `ssl.truststore.password` properties are required to access the CA certificate that we have just configured.

The consumer example is analogous and is omitted for brevity.

Interbroker encryption

The changes that were made so far focused on securing the communications between brokers and clients, leaving brokers to mingle using the regular cleartext protocol. Fortunately, configuring Kafka to use SSL for interbroker communications is a trivial matter. Edit `server.properties` and make the following one-line change.

```
inter.broker.listener.name=SSL
```

Restart the broker for the change to take effect. To verify that SSL is being used, check the startup logs. The server should output the current value of the `inter.broker.listener.name` property. You can also run `netstat` to ensure that there are no remaining connections on port `9092`, and that all traffic is now being served on port `9093`.

```
netstat -an | egrep "9092|9093"
```

```
tcp4    0   0  127.0.0.1.9093    127.0.0.1.65134   ESTABLISHED
tcp4    0   0  127.0.0.1.65134   127.0.0.1.9093    ESTABLISHED
tcp46   0   0  *.9093            *.*               LISTEN
tcp46   0   0  *.9092            *.*               LISTEN
```

The filtered output of `netstat` shows that, although both ports are listening, only `9093` is fielding an active connection.

Once it has been verified that SSL is working as planned, both for the client-broker and interbroker use cases, the recommended next step is to *disable the cleartext listener*. This is accomplished by updating all brokers and all clients first, allowing for both cleartext and SSL connections in the interim. Only once all clients and brokers have switched to the `SSL` listener, can the `PLAINTEXT` option be removed.

Disabling the cleartext listener can be done in one of two ways. The most straightforward approach is editing the static configuration in `server.properties` — removing the `PLAINTEXT` entry from both the `listeners` and `advertised.listeners` properties. The second way is to do it remotely, using per-broker or cluster-wide dynamic update modes. Dynamic configuration updates are discussed in Chapter 9: Broker Configuration.

A reader knowledgeable in cryptography may experience a slight confusion around Kafka's somewhat cavalier overloading of the term 'plaintext'. The `PLAINTEXT` protocol refers to a *cleartext* transmission, in that it deliberately avoids encryption. It is not an 'input to a cipher', as the term 'plaintext' might ordinarily imply.

Once the cleartext listener has been disabled, it is good practice to follow up with a corresponding firewall *deny* rule for all traffic inbound on port `9092`.

Broker-to-ZooKeeper encryption

While traffic between the Kafka brokers and the ZooKeeper ensemble can also be encrypted, the current version of Kafka (2.4.0 at the time of writing) does not support this feature natively. KIP-513[32] is targeting this for inclusion in release 2.5.0. To be clear, the limitation is specifically with the broker component of Kafka, not ZooKeeper. The latter supports mutual TLS.

[32]https://cwiki.apache.org/confluence/x/Cg6YBw

In the meantime, for the particularly security-minded deployments where encryption between brokers and the ensemble is a mandatory requirement, consider tunnelling the connection over a secure point-to-point link. This can be accomplished by positioning VPN terminators directly on the broker nodes, such that unencrypted traffic never leaves a broker. Alternatively, one can use a service mesh or a lightweight proxy capable of transparently initiating TLS connections, which are then natively terminated on ZooKeeper.

Encryption at rest

The encryption methods discussed earlier protect the confidentiality of data in transit. When data arrives at the broker, it will be persisted to its attached storage in indexed log segments — in cleartext.

Kafka does not have native facilities for enabling encrypted storage of record data. One has to resort to external options to accomplish this. The two popular approaches are full disk encryption and filesystem-level encryption. Both ensure that the disk is protected from unauthorized access when it is detached from the host.

While both forms of storage encryption provide a high level of protection against threats outside the host, the only way to protect the confidentiality of persisted data from embedded threats (rogue processes executing on the host) is to utilise end-to-end encryption. This involves encrypting outgoing records on the producer with either a shared-key or an asymmetric cipher — to be decrypted at the consumer end. This strategy ensures that neither the broker nor any intermediate conduits are aware of record contents. Kafka does not support end-to-end encryption natively. There is a provisional KIP-317[33] that discusses the prospect of adding this capability in the future. There are several open-source projects that implement this capability over the top of Kafka's serialization mechanism. One such example is the open-source *Kafka Encryption* project, hosted on GitHub at github.com/Quicksign/kafka-encryption[34].

When using end-to-end encryption, the information entropy of record batches approaches its theoretical maximum of unity. Therefore, it is best to disable compression, as the latter will only burn through CPU cycles without decreasing the payload size. (If anything, it will likely go up due to the overheads of compression.) For more information on compression, see Chapter 12: Batching and Compression.

End-to-end encryption, being focused solely on the record payload, does not eliminate the need for transport layer security. SSL (TLS) covers all aspects of the information exchange between clients and brokers, including metadata, record headers, offsets, group membership, and so on. SSL also protects the integrity of the data, guards against man-in-the-middle attacks, and uses X.509 certificates to provide assurance to the client party that the broker is authentic.

[33]https://cwiki.apache.org/confluence/x/AFIYBQ
[34]https://github.com/Quicksign/kafka-encryption

Authentication

Kafka supports several modes for attesting the identity of the connected clients. This section explores the authentication options at our disposal.

Mutual TLS

Mutual TLS (mTLS), also known as client-side X.509 authentication or two-way SSL/TLS, utilises the same principle of certificate signing used by conventional TLS, but in the opposite direction. In addition to the mandatory server-side authentication, the client presents a certificate that is verified by the broker. Each client will have a dedicated certificate, signed by a CA that is trusted by the broker. It may be the same CA used for the broker certificate signing, as is often the case.

To enable client authentication on the broker, one must set the `ssl.client.auth` property in `server.properties`. The permissible values are:

- `none`: Client authentication is disabled.
- `requested`: Client may optionally initiate SSL authentication, but this is not mandated by the broker. However, should the client present its certificate, it will be verified.
- `required`: The broker mandates the use of client-side SSL authentication. The client will not be allowed to connect without a valid certificate.

The `requested` option is a permissive 'halfway house' setting that enables the gradual migration to mTLS from unauthenticated TLS. To begin the transition, set all brokers to use `ssl.client.auth=requested`. This can be done statically — by editing each individual `server.properties` file and bouncing the brokers, or dynamically — via the admin API or the CLI tools. Once all brokers are running in permissive mode, begin the rollout of client updates. Once all clients have been verified to work stably with mTLS, upgrade the `ssl.client.auth` setting to `required`.

In order to use this authentication scheme, the client must be equipped with a dedicated private key, signed with a CA certificate that is trusted by the broker. The key and certificate should be installed in the `keystore.client.jks` file, alongside the existing `truststore.client.jks` file. The process below mimics the previous examples.

The following provides a worked example for enabling mutual TLS. Once again, the examples are going to use the `/tmp/kafka-ssl` directory. The `ca-key` and `ca-cert` files have been carried over from the previous examples.

Generate the private key

Generate a private key for the client application, using `localhost` as the value for the 'first and last name' attribute, leaving all others blank.

```
keytool -keystore client.keystore.jks -alias localhost \
    -validity 365 -genkey -keyalg RSA
```

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ↩
    ST=Unknown, C=Unknown correct?
  [no]:  y

Generating 2,048 bit RSA key pair and self-signed
certificate (SHA256withRSA) with a validity of 365 days
  for: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ↩
      ST=Unknown, C=Unknown
```

This leaves a `client.keystore.jks` file in the current directory.

Sign the client certificate

```
keytool -keystore client.keystore.jks -alias localhost \
    -certreq -file client-cert-req
```

This produces the client certificate signing request file — `client-cert-req`.

Next, we will action the signing request with the existing CA:

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in client-cert-req \
    -out client-cert-signed -days 365 -CAcreateserial
```

The result is the `client-cert-signed` file, ready to be imported into the client's keystore, along with the CA certificate.

Starting with the CA certificate:

```
keytool -keystore client.keystore.jks -alias CARoot \
    -import -file ca-cert
```

```
Enter keystore password:
Owner: CN=localhost
Issuer: CN=localhost
Serial number: 8f444e15ad8f7067
Valid from: Thu Jan 02 09:28:45 AEDT 2020 until: ▯
    Fri Jan 01 09:28:45 AEDT 2021
Certificate fingerprints:
   SHA1: 70:55:42:23:69:A1:EA:E8:13:49:41:CC:C3:CE:A3:▯
      7B:CB:25:F8:08
   SHA256: 7E:CC:21:57:5B:8C:FB:90:D9:9E:2B:84:76:C4:▯
      E1:83:D0:2D:B5:D1:17:3A:D2:D5:5A:4D:C5:CB:F3:9B:32:DD
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
Trust this certificate? [no]:  y
Certificate was added to keystore
```

Moving on to the signed certificate:

```
keytool -keystore client.keystore.jks -alias localhost \
    -import -file client-cert-signed
```

```
Enter keystore password:
Certificate reply was installed in keystore
```

Configure the broker to require authentication

Edit `server.properties`, adding the following line.

```
ssl.client.auth=required
```

Restart the server for the changes to take effect. To verify, you can run the producer client from the previous example. The connection should fail with the following error:

```
1:58:57/583   INFO  [kafka-producer-network-thread | producer-1]: 
    [Producer clientId=producer-1] Failed authentication with 
    localhost/127.0.0.1 (SSL handshake failed)
11:58:57/584   ERROR [kafka-producer-network-thread | producer-1]: 
    [Producer clientId=producer-1] Connection to node -1 
    (localhost/127.0.0.1:9093) failed authentication due to: SSL 
    handshake failed
```

That is to be expected; our SSL-enabled client has yet to be configured for client-side authentication.

Deploy the private key and signed certificate to the client

```
cp /tmp/kafka-ssl/client.keystore.jks \
    ~/code/effectivekafka
```

Configure the client to provide authentication

Enabling mutual authentication on the client requires the addition of several properties — namely, the SSL keystore configuration. In the snippet below, we have provided the `ssl.keystore.location`, `ssl.keystore.password` and `ssl.key.password` — being the mirror image of the server-side configuration.

```java
final var config = new HashMap<String, Object>();
config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
           "localhost:9093");
config.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
           "SSL");
config.put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
           "https");
config.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
           "client.truststore.jks");
config.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,
           "secret");
config.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG,
           "client.keystore.jks");
config.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG,
           "secret");
config.put(SslConfigs.SSL_KEY_PASSWORD_CONFIG,
           "secret");
config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
           StringSerializer.class.getName());
config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
           StringSerializer.class.getName());
```

```
config.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
           true);

// ... the existing producer code
```

Identifying the user principal

When authenticating a client to the broker, hostname verification is not performed. Instead, the user principal is taken verbatim as the value of the CN attribute of the certificate. One can change how a certificate's attributes translate to a username, by overriding the `ssl.principal.mapping.rules` property to provide an alternate mapping. The format of this setting is a list, where each rule starts with the prefix `RULE:`, and specifies a regular expression to match the desired attributes, followed by a `/` (slash) character, then by a replacement, another slash, and finally, by an optional `U` or `L` character indicating whether the replacement should be capitalised or forced to lower case, respectively. The format is depicted below.

```
RULE:pattern/replacement/
RULE:pattern/replacement/[LU]
```

There are several examples listed on the official Kafka documentation page at kafka.apache.org/documentation[35].

Mutual TLS is frequently used in machine-to-machine authentication — ideal for the typical Kafka use case, where the client party is an application that operates independently of its end-users. There may be complications, however. The degree of authentication assurance in mTLS is limited by the level of rigour present in the certificate signing process. Specifically, the broker can determine with a high level of assurance that the client is a *trusted party*. But which trusted party? That is the more challenging question.

Recall, the client is trusted because it presents a valid certificate signed by a trusted issuer. The username implied by the certificate (represented by the CN by default) is assumed to be strongly associated with the party that presented the certificate, because this relationship was presumably independently verified by the issuer — a CA that we trust *a priori*.

Consider, for example, two trusted clients: *C0* and *C1*. Both have a dedicated private key, signed by a trusted CA for use with a specific CN. However, if the signing process is naively implemented, there is little stopping *C0* from issuing another signing request to the CA, using the username of *C1* in the CN field. If this were to happen, *C0* could connect to the broker with an alternate certificate — impersonating *C1*. In order to prevent this from occurring, the process of signing certificates cannot be left entirely in the clients' hands — irrespective of whether the clients operate within the organisation's perimeter. The relationship between a client and the CA must be individually authenticated, so that a client can only issue a signing request for a CN that has been authorized for that client.

[35]https://kafka.apache.org/documentation/#security_authz_ssl

> A layered approach to information security — the use of principles such as *Defence in Depth* — imply that an organisation should trust internal entities little more than it should trust external ones. Recent studies have indicated that as much as 60% of attacks originate from within the perimeter. And while PKI and public-key cryptography allow us to reason concretely as to the authenticity or the integrity of data, they are predicated on assumptions that are often taken for granted.

Another challenge of using mTLS is that the identity of clients is not known in advance, as there is no requirement to explicitly enrol each user principal into Kafka. On one hand, this clearly simplifies identity generation by decoupling it from the authentication mechanism. On the other hand, the management of identities is significantly more complicated than username/password-based methods, as Kafka does not support certificate revocation. In other words, if the private key of one client (or a small group of clients) becomes compromised, there is no way to instruct the brokers to blacklist the offending certificates. At best, one would have to deploy a new CA certificate in place of the compromised one and sign all remaining legitimate client certificates with the novated CA.

ℹ Technically, although certificates cannot be revoked, one can remove all privileges from the affected user principals, and require the rotation of the affected usernames. Frankly, this cannot be called a definitive solution — more of a workaround. Although username rotation fixes the immediate threat, it does so at the authorization layer, which is distinct from authentication. There is still a threat that the original user's permissions might be accidentally restored later, at which point the compromised key will become a threat again. Without regenerating the CA certificate, there is no way to *permanently* disable a user.

⚠ Certificate revocation has been an open issue in Kafka since May 2016. As such, avoid the use of mutual TLS in environments where client deployments are not secure, and where there is a likelihood of a key compromise — even if the client acts as a low-privilege user on a test system. While the implication a single compromise might not be catastrophic, the liquidation of the consequences will be long and laborious. To that point, when using mTLS with multiple Kafka clusters (e.g. multiple environments or geographical sites), it is best to segregate trust chains, so that each cluster trusts its own intermediate CA certificate but not the others', and clients are selectively issued with certificates that are individually signed for the specific clusters that they are legitimately expected to access.

Mutual TLS is used frequently for authenticating clients to managed Kafka providers. The service consumer will typically provide an intermediate CA certificate to the service provider, which then allows the service consumer to deploy any number of Kafka clients with dedicated certificates.

ℹ Despite operating at different layers of the OSI model and being independent in theory, two-way SSL cannot be used in conjunction with application-level authentication schemes. In Kafka, it is either one or the other.

SASL

Kafka supports *Simple Authentication and Security Layer* (SASL) — an extensible framework for embedding authentication in application protocols. The strength of SASL is that it decouples authentication concerns from application protocols, in theory allowing any authentication mechanism supported by SASL to be embedded in any application that uses SASL. SASL is not responsible for the confidentiality of the message exchange, and is therefore rarely used on its own. The most common deployment model of SASL involves TLS.

GSSAPI

Kafka supports the Generic Security Service API (GSSAPI), which enables compliant security-service vendors to exchange opaque tokens with participating applications. These tokens are internally tamper-proof, and are used to establish a security context among collaborating parties. Once the context is established, the parties can communicate securely and access resources.

> GSSAPI is commonly associated with Kerberos, being its dominant implementation. The primary distinction is that the Kerberos API has not been standardised among vendors, whereas GSSAPI has been vendor-neutral from the outset. In a manner of speaking, GSSAPI standardises Kerberos.

The official Kafka documentation treats GSSAPI synonymously with Kerberos. The professed relationship between GSSAPI and Kerberos typically extends to centralised directory services — Active Directory (predominantly due to its native Kerberos version 5 support) and similarly positioned corporate single sign-on (SSO) services.

Kerberos and Active Directory work best for interactive users in a corporate setting; however, Kafka users are rarely individuals, but applications. Even when a Kafka client acts on behalf of an end-user, it will ordinarily authenticate itself to the broker via an end-user-agnostic mechanism — typically a *service account*. The authenticated session outlives any single user interaction; due to their time and resource costs, it is generally infeasible to spawn a new set of connections for every user interaction, only to tear them down shortly afterwards. In the overwhelming majority of cases, service accounts are the only practical way to manage Kafka clients.

In theory, a service account could be provisioned in a centralised corporate authentication server such as Active Directory. On the surface, it even sounds beneficial. By centralising service accounts, it should make their governance a more straightforward and transparent affair.

While the theoretical advantages are not disputed, the main challenge with the centralised model is that it only works if the principal in question can be modelled as a resource in a directory and all the resources it consumes are represented in some sort of a permission model that can be associated with the principal. In other words, an application can access all of its resources using a single set of credentials. (Or these credentials are somehow related, and their relationship can be modelled accordingly.) In reality, backend application components may consume numerous disparate resources, not all of which may be enrolled in the directory. For example, a single application may connect to Kafka, a Postgres database, a Redis cluster, and may consume third-party APIs that are external to the organisation. While Kafka and Postgres support Kerberos, Redis does not. And a third-party API will almost certainly not support Kerberos. (Some service providers may indirectly support Kerberos, typically via a federation protocol.)

When the time comes to control the permissions of a service account, the administrator might manipulate the representation of the service account in the directory. Perhaps the service account should be disabled altogether, which is a trivial 'one-click' action in a directory service such as Active Directory. This creates a false sense of security, as the permissions for the service account do not cover all resources; while the service account may appear to be disabled in the centralised directory, the underlying principal is still able to operate on a subset of its resources.

It is a common misunderstanding that centralised authentication systems exist to simplify the on-boarding process. In practice, the benefits of such systems are predominantly concentrated in the off-boarding scenario. It is convenient to quickly set up a profile for a new employee with one click of a button, granting them access to all necessary systems. Convenient — but not essential. Without a centralised authentication system, the on-boarding process may take days or weeks to complete. And while this is not ideal, it is mostly the productivity loss that affects the organisation — a cost that can be quantified and managed accordingly. Conversely, termination of employment requires immediate action from a security standpoint — revoking the employee's access simultaneously from all resources previously available to them, preferably before the person has left the premises. Failure to act in a timely manner, or forgetting to apply the change to some resources, may pose an immediate and persistent threat to the organisation, particularly if the employee was not completely content with the circumstances of their dismissal.

While the benefits of centralised authentication are clear for end-users and administrators, particularly in a corporate setting, the equivalent benefits cannot be easily reproduced for service accounts — not unless the organisation mandates that all technical infrastructure supports Kerberos and all interactions between application components and their dependencies are authenticated accordingly. Stated plainly, this is a pipe dream.

For service accounts and similar integration use cases, the prevalent industry trend is in the migration away from centralised directory-based authentication systems towards centralised secrets management systems. In other words, rather than managing principals, which we established is futile, we manage their credentials and any other secret material they need to function. Provided the principal receives the entirety of its credentials from the secrets management system and does not cache them locally, then disabling the principal will *eventually* affect its ability to consume downstream resources, effectively isolating it.

One notable drawback of this scheme is that the effect of disabling a principal may not be timely. While it makes future retrieval of secret material impossible, it does not necessarily render the existing material invalid or obsolete. On top of this, the client may have cached the secret material or the latter may have become compromised by other means. This is why security best-practices suggest *frequent rotation of secret material*, down to the order of minutes — the time-exposure of a sensitive artifact is thereby minimised.

This chapter does not delve further into the authentication cases for GSSAPI, not solely because of the statements above. Rather, the benefit of demonstrating Kerberos is not worth the complexity of setting it up. It is also adequately documented at kafka.apache.org/documentation[36]. Instead, the focus will shift towards other SASL authentication methods that are less draconian, more straightforward to administer, and are likely to remain relevant to the reader for many years to come.

PLAIN and SCRAM

The complete source code listings for the SASL producer and consumer client examples are available at github.com/ekoutanov/effectivekafka[37] in the `src/main/java/effectivekafka/sasl` directory.

SASL offers two username/password-based authentication modes: PLAIN and SCRAM. (There are more in existence, but Kafka just supports these two for authenticating clients.) The former refers to cleartext authentication where credentials are presented verbatim. The latter is an acronym for *Salted Challenge Response Authentication Mechanism* — a protocol designed to fulfil authentication without the explicit transfer of credentials.

To the user, there is little discernible difference between the two options. In both cases, the application is configured with a username and password pair, with the only difference being the fully-qualified class name of the JAAS (Java Authentication and Authorization Service) module. Under the hood, the two are very different. PLAIN provides no confidentiality of its own, requiring the use of an encrypted channel. SCRAM is secure in its own right, providing confidentiality in the absence of any transport layer encryption.

In case the question might arise, PLAIN is not an acronym. The exact origin of the name is unspecified, but it is likely a candid capitalisation of the adjective 'plain' — being "simple or basic in character", as opposed to an abbreviation of 'plaintext' — the cryptographic term meaning "input to a cipher". Indeed, RFC 4616[a] specifies PLAIN as a *simple mechanism for the exchange of passwords*. The exchange always occurs in cleartext — there is no intention of encryption at the application level. As such, PLAIN is intended to be used in concert with lower-level mechanisms that guarantee the confidentiality of data. Quoting from the RFC:

> As the PLAIN mechanism itself provides no integrity or confidentiality protections, it

[36]https://kafka.apache.org/documentation/#security_sasl_kerberos
[37]https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/sasl

should not be used without adequate external data security protection, such as TLS services provided by many application-layer protocols. By default, implementations **should not** advertise and **should not** make use of the PLAIN mechanism unless adequate data security services are in place.

[a] https://tools.ietf.org/html/rfc4616

Guided by the *Defence in Depth* principle, the SCRAM option should be preferred over PLAIN. This ensures that no control will singlehandedly impact the integrity of the system if compromised. SCRAM acts in both directions: not only must the client prove to the broker that it has the password, but the broker is required to do the same. (Caveat below.) In other words, SCRAM protects the client from connecting to a rogue broker, which acts in addition to the certificate-based attestation mechanism used natively in TLS. The only practical drawback of SCRAM over PLAIN is the added network round trip, which is necessary for both parties to identify one another. (SCRAM requires a total of two round-trips.) However, this penalty is paid once, during connection establishment — it has no subsequent bearing on throughput or latency.

Although SCRAM authentication is bidirectional, it is not completely symmetric — there is a subtle difference between the assurance each party provides to the other. Specifically, the client must prove to the broker that it has *present* knowledge of the password — at the time of connection establishment; whereas the broker only needs to prove that it knew the password at some point in time. This is a desirable property — it relieves the broker from having to persist the password verbatim, and therefore risking exposure; instead, the broker persists irreversible derivations of the password that are subsequently used for mutual authentication. In theory, the client can discard the password and store the hashed version, provided it knows the salt used on the server (which is disclosed as part of the authentication flow).

Because of the similarity in how the two authentication methods appear to the user, the worked examples will focus on the SCRAM case.

Configure the broker to use SASL/SCRAM

We need to create a new listener that supports the use of SASL authentication. Edit `server.properties`, ensuring that it is in line with the following:

```
listeners=PLAINTEXT://:9092,SSL://:9093,SASL_SSL://:9094
advertised.listeners=PLAINTEXT://localhost:9092,\
    SSL://localhost:9093,SASL_SSL://localhost:9094
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,\
    SSL:SSL,SASL_PLAINTEXT:SASL_PLAINTEXT,SASL_SSL:SASL_SSL
sasl.enabled.mechanisms=SCRAM-SHA-512
listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule \
    required;
```

Having saved the file, restart the broker for the change to take effect.

Working through the above file, line-by-line:

- We added another listener named `SASL_SSL`, bound to port `9094`.
- The listener has been advertised as `SASL_SSL://localhost:9094`.
- The security protocol map has been left as is, given it already contains an `SASL_SSL:SASL_SSL` mapping out of the box.
- The enabled SASL mechanism is `SCRAM-SHA-512`.
- The `listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config` property provides a minimal JAAS document that mandates the use of the `ScramLoginModule`.

SASL authentication is incompatible with SSL client authentication. When the `ssl.client.auth` property is set to `requested` or `required`, the SSL authentication settings will only be applied to the `SSL` listener (or any listener mapped to the `SSL` security protocol). When connecting over the `SASL_SSL` listener, the SSL client authentication settings will be ignored. It would have been advantageous to run SASL on top of SSL client authentication for added security; and while this is possible in theory, Kafka does not support the conjunction of the two. The reason is that Kafka extracts the user principal from the attributes of the client certificate, which may conflict with the username provided over SASL.

Kafka supports `SCRAM-SHA-256` and `SCRAM-SHA-512`. Both can be enabled if need be, by setting the `sasl.enabled.mechanisms` broker property to `SCRAM-SHA-256,SCRAM-SHA-512`. Security-wise, both SHA-256 and SHA-512 are generally considered to be very strong, with the latter having better collision resistance. Different hardware favours different functions, with SHA-512 optimised for use on 64-bit processors; SHA-256 being more performant on 32-bit processors. SHA-256 remains the more common choice for the time being.

Provision the user

Kafka's SCRAM implementation uses ZooKeeper as a credential store — for persisting usernames and hashed passwords (which are also salted). Cleartext passwords are never persisted, but the hashes are world-readable. This implies that ZooKeeper should be deployed on a highly trusted network that is protected from access by unauthorised parties. Configuration is administered directly against ZooKeeper, using the `kafka-configs.sh` CLI. In the following example, we will provision a new user with the username `alice`, the password being `alice-secret`.

```
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --alter \
    --add-config 'SCRAM-SHA-512=[password=alice-secret]' \
    --entity-type users --entity-name alice
```

```
Completed Updating config for entity: user-principal 'alice'.
```

When enrolling credentials into ZooKeeper, they need to be specified separately for each variation of the SCRAM algorithm. In other words, `SCRAM-SHA-256` credentials may differ from `SCRAM-SHA-512`, and can be administered independently. The `kafka-configs.sh` CLI conveniently supports specifying lists of key-value pairs in the `--add-config` flag.

To list the current configuration for a user, run the following (replacing the username as appropriate):

```
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --describe --entity-type users --entity-name alice
```

This will output the current configuration:

```
Configs for user-principal 'alice' are SCRAM-SHA-512=salt= □
    ZWhuNHk2b2ZxaHRxcHQxamlhdDYzbzg=,stored_key=hpRpfIoJZc □
    Orhy9/6Fh80VsBhNCKKpMTZtWKkCs08H8us15pphr5upfTGjvGYYON □
    EeDfw002WUqfT6g+TRJqA==,server_key=y8yhtDe201H2hK9a6tk □
    eidFTL2A3E4KV+Pd6U5QfXc5ndj0nzzf15N1xX0m5W4EVccDphXIDG □
    T5wtuEAAnmv5g==,iterations=4096
```

Note that the output of `--describe` is different to what was fed as the input to `--add-config`. This is because SCRAM persists derived values of the password. These values can be used to cryptographically verify the mutual knowledge of the password during an authentication session, but the original password cannot be recovered using this method.

Credentials may be deleted using the `--delete-config` flag:

```
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --alter \
    --delete-config 'SCRAM-SHA-512'\
    --entity-type users --entity-name alice
```

Configure the client

Configuring a client to use SASL requires just a few additions to the configuration map. The example below is a fully-functional producer that uses a combination of SSL and SASL.

```
import static java.lang.System.*;

import java.util.*;

import org.apache.kafka.clients.*;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.config.*;
import org.apache.kafka.common.security.scram.*;
import org.apache.kafka.common.serialization.*;

public final class SaslSslProducerSample {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "getting-started";

    final var loginModuleClass = ScramLoginModule.class.getName();
    final var saslJaasConfig = loginModuleClass
        + " required\n"
        + "username=\"alice\"\n"
        + "password=\"alice-secret\";";

    final Map<String, Object> config = Map
        .of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9094",
            CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
            "SASL_SSL",
            SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
            "https",
            SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
            "client.truststore.jks",
            SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,
            "secret",
            SaslConfigs.SASL_MECHANISM,
            "SCRAM-SHA-512",
            SaslConfigs.SASL_JAAS_CONFIG,
            saslJaasConfig,
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName(),
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName(),
            ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
            true);

    try (var producer = new KafkaProducer<String, String>(config)) {
      while (true) {
        final var key = "myKey";
        final var value = new Date().toString();
        out.format("Publishing record with value %s%n",
                   value);

        final Callback callback = (metadata, exception) -> {
          out.format("Published with metadata: %s, error: %s%n",
                     metadata, exception);
        };

        // publish the record, handling the metadata in the callback
        producer.send(new ProducerRecord<>(topic, key, value),
                      callback);

        // wait a second before publishing another
        Thread.sleep(1000);
      }
    }
  }
}
```

The differences from the pure-SSL example are in the following:

- Connection to a different port. In this case, we are connecting to port `9094`.
- Use of `SASL_SSL` for the `security.protocol` setting.
- Addition of the `sasl.mechanism` property, set to `SCRAM-SHA-512`.
- Specifying the JAAS configuration via the `sasl.jaas.config` property.

The JAAS configuration for example above is:

```
org.apache.kafka.common.security.scram.ScramLoginModule required
    username="alice"
    password="alice-secret";
```

Interbroker authentication

Although we changed client authentication to `SASL_SSL`, the interbroker listener remained `SSL` from the first example. We can upgrade interbroker communications to use authentication in addition to encryption; however, this requires us to configure a set of credentials specifically for broker use.

Start by creating a set of admin credentials (replace `admin` and `admin-secret` as appropriate):

```
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --alter \
    --add-config 'SCRAM-SHA-512=[password=admin-secret]' \
    --entity-type users --entity-name admin
```

Then edit `server.properties` to line up with the following. (Only the changed lines are shown.)

```
inter.broker.listener.name=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule \
        required \
        username="admin" \
        password="admin-secret";
```

This is where we need to be careful. Having specified credentials in `server.properties`, we *should* harden the file permissions to prevent the file from being read by anyone other than the currently logged in user. If this user is different from the user that will run Kafka, then the ownership and permissions should be adjusted accordingly.

```
chmod 600 $KAFKA_HOME/config/server.properties
```

Having saved the file and adjusted its permissions, restart the broker for the change to take effect.

Summary of the changes:

- The interbroker listener was changed to `SASL_SSL`.
- The SASL mechanism for interbroker use was assigned via the `sasl.mechanism.inter.broker.protocol` property.
- The JAAS configuration defined in `listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config` was expanded to include a username and password pair.

Having switched the interbroker protocol, run `netstat` to verify that interbroker connections are directed at port `9094`:

```
netstat -an | egrep "9092|9093|9094"
```

```
tcp4    0   0  127.0.0.1.9094    127.0.0.1.56410   ESTABLISHED
tcp4    0   0  127.0.0.1.56410   127.0.0.1.9094    ESTABLISHED
tcp46   0   0  *.9094            *.*               LISTEN
tcp46   0   0  *.9093            *.*               LISTEN
tcp46   0   0  *.9092            *.*               LISTEN
```

Supplying an external JAAS configuration

In previous SASL examples, we supplied the JAAS configuration in-line, via a configuration property. And while this approach is convenient, it may not always be appropriate — depending on whether the application supports this method of configuration. Where in-line configuration is not supported, JAAS can be configured using an external file.

To alter the previous example to use an external JAAS configuration file, edit `server.properties` and comment out (or remove) the `listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config` property. Save the file.

If you attempt to start the broker in this state, it will bail out with the following error:

```
[2020-01-02 19:22:25,890] ERROR [KafkaServer id=0] Fatal error ⏎
    during KafkaServer startup. Prepare to shutdown ⏎
    (kafka.server.KafkaServer) ⏎
    java.lang.IllegalArgumentException: Could not find a ⏎
    'KafkaServer' or 'sasl_ssl.KafkaServer' entry in the ⏎
    JAAS configuration. System property ⏎
    'java.security.auth.login.config' is not set
```

To fix this error, create a `kafka_server_jaas.conf` file in `$KAFKA_HOME/config`, with the following contents:

```
sasl_ssl.KafkaServer {
    org.apache.kafka.common.security.scram.ScramLoginModule
        required
        username="admin"
        password="admin-secret";
};
```

The JAAS configuration contains sensitive information. Ensure it is not readable by other users by running the following:

```
chmod 600 $KAFKA_HOME/config/kafka_server_jaas.conf
```

When starting the Kafka broker, it needs to be told where to find the JAAS configuration. This can be accomplished by setting the `java.security.auth.login.config` system property, which is passed via the `KAFKA_OPTS` environment variable, as shown below.

```
JAAS_CONFIG=$KAFKA_HOME/config/kafka_server_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/kafka-server-start.sh \
    $KAFKA_HOME/config/server.properties
```

Switching to SASL/PLAIN

It was previously stated that SASL/PLAIN and SASL/SCRAM are not materially differentiated from a user's perspective. To change clients over to SASL/PLAIN, set the `sasl.mechanism` client property to `PLAIN` and replace the fully-qualified class name of the JAAS module from `org.apache.kafka.common.security.s` to `org.apache.kafka.common.security.plain.PlainLoginModule`.

On the broker, the differences are slightly more perceptible. PLAIN mode does not use ZooKeeper to store credentials. Instead, credentials are defined directly in the JAAS configuration.

To enable SASL/PLAIN, edit `server.properties`, changing the following lines:

```
sasl.enabled.mechanisms=PLAIN
sasl.mechanism.inter.broker.protocol=PLAIN
listener.name.sasl_ssl.plain.sasl.jaas.config= \
    org.apache.kafka.common.security.plain.PlainLoginModule
        required \
        username="admin" \
        password="admin-secret" \
        user_admin="admin-secret" \
        user_alice="alice-secret";
```

Also, remove the `listener.name.sasl_ssl.scram-sha-512.sasl.jaas.config` property from the SASL/SCRAM example. (Unless you need to maintain SCRAM alongside PLAIN.)

On the client-side, change the value of the `sasl.mechanism` property to `PLAIN`.

OAuth bearer

The `OAUTHBEARER` SASL mechanism enables the use of OAuth 2.0 Access Tokens to authenticate user principals. *The primary motivation for OAuth support is not for security*, but for ease of integration and testing, allowing applications to impersonate users by way of an *Unsecured JWS* token in non-production environments. In other words, an application can pose as an arbitrary user by issuing an unsigned JWT (JSON Web Token), where the header does not specify an algorithm:

```
{
  "alg": "none"
}
```

The JWS Signature used for unsecured tokens is an empty octet string, which base-64 encodes simply to an empty string.

Unsecured tokens function out-of-the-box with minimal configuration and with no OAuth 2.0 infrastructure required. The principal's username is taken directly from the `sub` (subject) claim in the JWT.

In its default guise, the `OAUTHBEARER` mechanism cannot be used securely in production environments, as it cannot verify user claims. In order to productionise this algorithm, one must implement a pair of callback handlers. The callbacks are necessary to allow the client to generate and sign access tokens, and for the broker to validate them against a trusted certificate.

On the client, implement the `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler` interface to retrieve a token. The implementation must be able to handle an instance of an `OAuthBearerTokenCallbac` feeding it a well-formed `OAuthBearerToken`. The implementation class is specified via the `sasl.login.callback.hand` client property.

On the broker, implement the `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler` interface capable of handling an `OAuthBearerValidatorCallback`. Its role is to validate the attributes of the token and feed a validated token to the callback. The implementation class is configured via the `listener.name.sasl_ssl.oauthbearer.sasl.server.callback.handler.class` property. The class definition, along with all dependencies must be on the classpath of the broker's JVM instance. This can be accomplished by packaging the callback handler and its dependencies into a Jar file, and placing the latter into `$KAFKA_HOME/libs`.

Rather than implementing OAuth handling from scratch, you can use the open-source *Kafka OAuth* project, located at github.com/jairsjunior/kafka-oauth[38].

A secondary objective of supporting OAuth bearer tokens was the internal testing of SASL. Kafka's SASL implementation can now be considered mature, but nonetheless, the `OAUTHBEARER` can still be useful for testing with different users.

Delegation tokens

Concluding the discussion on authentication methods, we have *delegation tokens*. Delegation tokens are used as a lightweight authentication mechanism that is complementary to SASL. It was introduced in KIP-48[39] as part of release 1.1.0. The initial motivation for delegation tokens was to simplify

[38]https://github.com/jairsjunior/kafka-oauth
[39]https://cwiki.apache.org/confluence/x/tfmnAw

the logistics of key distribution for Kerberos clients — specifically, the need to deploy TGTs or keytab files to each client. Delegation tokens also reduce the overhead of the authentication process — rather than engaging the KDC to get a ticket and periodically renewing the TGT, the authentication protocol is reduced to verifying the delegation token, which is persisted in ZooKeeper. This approach is also more secure, reducing the blast radius of compromised credentials — if a delegation token is covertly obtained, the attacker is limited to the permissions attached to the delegation token. By comparison, the compromise of a Kerberos TGT or keytab carries more dire consequences.

The most common use case for delegation tokens is event stream processing where a swarm of worker nodes is orchestrated by a centralised coordinator. Worker nodes tend to be ephemeral. Rather than provisioning access independently for each worker node, the coordinator creates a time-bounded delegation token, then spins up a worker node, handing it the newly-generated token. Under this scheme, only the coordinator node requires access to long-lived credentials (a TGT or keytab for Kerberos, or a username/password pair for SCRAM or PLAIN); the highly sensitive material never leaves the coordinator.

Delegation tokens are issued for a finite lifespan — an upper bound on the maximum age of a token. Furthermore, a token has an expiration time, which sets a soft limit on the time that the token may be used to authenticate a client. The expiry time may be extended by renewing the token, subject to the upper bound enforced by its lifespan. An expired token is eventually purged from ZooKeeper using a background housekeeping thread. Because purging happens asynchronously, it is possible for a token to be used for some time after its official expiry.

Enable delegation tokens on the broker

To enable delegation tokens, edit `server.properties`, adding the following lines:

```
delegation.token.master.key=secret-master-key
delegation.token.expiry.time.ms=3600000
delegation.token.max.lifetime.ms=7200000
```

The `delegation.token.master.key` property specifies a key for signing tokens that must be shared by all brokers in a cluster. This property is required — delegation tokens are disabled without it. Replace the string `secret-master-key` as appropriate.

The `delegation.token.expiry.time.ms` specifies the token expiry time in milliseconds. The default value is `86400000` (24 hours).

The `delegation.token.max.lifetime.ms` specifies the hard upper bound on the lifespan of the token in milliseconds. When issuing a token, the user can specify any value for the `--max-life-time-period` that is below this setting. The default value of `delegation.token.max.lifetime.ms` is `604800000` (7 days).

Restart the broker for the changes to take effect.

Creating delegation tokens

To create a delegation token, use the `kafka-delegation-tokens.sh` CLI, as shown in the example below.

```
$KAFKA_HOME/bin/kafka-delegation-tokens.sh \
    --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties \
    --create --max-life-time-period -1 \
    --renewer-principal User:admin
```

The owner of the token is the principal specified in the supplied properties file. In our example, `client.properties` has been configured with the credentials of the `admin` user. This is not ideal in production, as it gives the token bearer significantly more privileges than it likely requires.

Running the above command produces the following output:

```
Calling create token operation with renewers : □
    [User:admin] , max-life-time-period :-1
Created delegation token with tokenId : □
    doFHaIYjQwWhyeBZrBW54w

TOKENID
doFHaIYjQwWhyeBZrBW54w
 □
HMAC
AFtqnd/cV/fFCawRYhPIHqe9sLZGegYscYu1o8BwSDn11 □
    rIqjKWRsKyLS9+CJ5Jor4RsAckTcS0IAWvdAlPqjQ==
 □
OWNER                   RENEWERS
User:admin              [User:admin]
 □
ISSUEDATE               EXPIRYDATE              MAXDATE
2020-01-02T11:43        2020-01-01T12:43        2020-01-03T11:43
```

The `--max-life-time-period` flag specified the maximum lifespan of the token in milliseconds. If set to `-1`, the maximum admissible value specified by the `delegation.token.max.lifetime.ms` broker property is assumed.

The `--renewer-principal` flag specifies the user who is allowed to renew the token. This user may be different from the owner. When renewing the token, the user must present their credentials in addition to the token's HMAC value.

To list delegation tokens, invoke `kafka-delegation-tokens.sh` with the `--describe` switch:

```
$KAFKA_HOME/bin/kafka-delegation-tokens.sh \
    --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties \
    --describe
```

Kafka presently does not allow one user to create a delegation token on behalf of another (where the owner principal is different to the maker of the token). KIP-373[40] is slated to address this shortfall.

Configuring clients

Delegation tokens are passed in similarly to credentials, using the SASL/SCRAM method. The `username` attribute is set to the token ID, while the `password` attribute is assigned the `HMAC` value from the table above. In addition to these attributes, a third attribute `tokenauth` must be present, set to `true`. This attribute distinguishes token authentication from conventional username/password authentication.

```
final var saslJaasConfig = loginModuleClass
    + " required\n"
    + "username=\"doFHaIYjQwWhyeBZrBW54w\"\n"
    + "password=\"AFtqnd/cV/fFCawR...PqjQ==\"\n"
    + "tokenauth=\"true\";";
```

Renewing tokens

The renewal of a token can only be attempted by the user listed in `--renewer-principal`, and must happen before the token expires. The `kafka-delegation-tokens.sh` command requires both the user's credentials and the token being renewed — specifically, the `HMAC` value:

```
$KAFKA_HOME/bin/kafka-delegation-tokens.sh \
    --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties \
    --renew --renew-time-period -1 \
    --hmac AFtqnd/cV/fFCawR...PqjQ==
```

The `--renew-time-period` flag specifies the duration of time (in milliseconds) that the token should be extended by. If set to `-1`, the token will be renewed for the duration specified by the `delegation.token.expiry.tim` broker property.

Expiring tokens

The expiration time of a token can be adjusted following its creation. The most common case is to explicitly invalidate a token, accomplished using the command below.

[40] https://cwiki.apache.org/confluence/x/cwOQBQ

```
$KAFKA_HOME/bin/kafka-delegation-tokens.sh \
    --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties \
    --expire --expiry-time-period -1 \
    --hmac AFtqnd/cV/fFCawR...PqjQ==
```

The `--expiry-time-period` switch specifies an extension on the expiration time in milliseconds. If set to `-1`, the token will be expired immediately.

Rotating secrets

Kafka does not yet have an elegant mechanism for rotating the shared secret. This is currently a three-step process:

1. Expire all existing tokens.
2. Perform a rolling bounce of the cluster with the new secret.
3. Generate and distribute new tokens.

As tokens are verified only during connection establishment, any clients that are already connected will continue to function normally. New connections using old tokens will be rejected, as will any attempt to renew or expire tokens.

ZooKeeper authentication

The final stretch in our authentication journey is hardening the link between the brokers and the ZooKeeper ensemble. ZooKeeper supports SASL client authentication using the DIGEST-MD5 method. (In our context, the client is the Kafka broker or a CLI tool.)

ZooKeeper authentication powers its authorization model, which is implemented by way of an Access Control List (ACL). An ACL applies individually to each znode, in a manner that is similar to UNIX file and directory permissions.

ZooKeeper supports the following permission types on znodes:

- `CREATE`: create a child node.
- `READ`: get data from a node and list its children.
- `WRITE`: set data for a node.
- `DELETE`: delete a child node.
- `ADMIN`: set permissions.

Permissions are recorded against a znode in a list. Each element is a triplet, comprising the authentication scheme, the allowed principal, and the set of permissions. There are several built-in schemes. The two notable ones that apply in our context are `world` — meaning anyone (including unauthenticated users), and `sasl` — referring to a user that has been authenticated via SASL.

The version of ZooKeeper (3.5.6) bundled with the latest Kafka version (2.4.0 at the time of writing) does not enforce authentication — only authorization. A client can connect without presenting any credentials — its session will be associated with the `world` scheme. Although we can't block unauthenticated connections, we can limit their level of access using znode ACLs. In effect, this is how ZooKeeper restricts unauthenticated connections — it accepts the connection but restricts their further actions.

Enabling authentication on a ZooKeeper ensemble that has already been confined to a segregated network seems excessive. And in some ways, it is. The security of information assets can be compounded almost indefinitely until the usability of the system is crippled for legitimate users. At which point one asks: *Have we just made the system less secure by blurring the line between legitimate and illegitimate users?*

The answer to this is subjective, and it really depends on several factors, such as existing security controls, the nature of the industry and level of regulation, existing organisational security policies and guidelines, the likelihood of a breach and the cost of mitigation. The following subsections include worked examples for enabling ZooKeeper authentication for new and existing clusters; however, the rest of the chapter and the book will assume that ZooKeeper authentication is disabled.

Enable authentication on ZooKeeper

ZooKeeper supports a different set of SASL methods compared to what Kafka offers its clients; nonetheless, the basic principles are the same. Like Kafka, ZooKeeper uses JAAS for configuration.

Create a new file named `zookeeper_jaas.conf` in the `$KAFKA_HOME/config` directory, with the contents below. The admin username will be `zkadmin`. Replace the value `zkadmin-secret` with a secure password for authenticating to ZooKeeper.

```
Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
        user_zkadmin="zkadmin-secret";
};
```

Make sure the file is not world-readable:

```
chmod 600 $KAFKA_HOME/config/zookeeper_jaas.conf
```

Edit `zookeeper.properties`, adding the following configuration:

```
authProvider.1=\
    org.apache.zookeeper.server.auth.SASLAuthenticationProvider
jaasLoginRenew=3600000
```

The location of the JAAS file can be specified using the `java.security.auth.login.config` system property, passed using the `KAFKA_OPTS` environment variable. (Both Kafka and ZooKeeper use the same wrapper scripts, which work off the same environment variables.) Restart ZooKeeper, but don't connect to it just yet.

```
JAAS_CONFIG=$KAFKA_HOME/config/zookeeper_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/zookeeper-server-start.sh \
    $KAFKA_HOME/config/zookeeper.properties
```

The next step is to author a client JAAS file that can be used by Kafka as well as CLI tools to connect to ZooKeeper. Create a file named `kafka_server_jaas.conf` in the `$KAFKA_HOME/config` directory, containing the following:

```
Client {
    org.apache.zookeeper.server.auth.DigestLoginModule required
        username="zkadmin"
        password="zkadmin-secret";
};
```

Make sure the file is not group-readable:

```
chmod 600 $KAFKA_HOME/config/kafka_server_jaas.conf
```

After ZooKeeper starts with the server-side JAAS configuration, connections will be authenticated using SASL and clients will be able to set ACLs on znodes as needed. However, any existing znodes that were created *prior* to enabling SASL will remain fully-accessible by the `world` scheme. Recall, ZooKeeper's authentication exists to support its authorization controls. Enabling authentication on its own has no effect on existing znodes. Fortunately, Kafka provides a convenient migration script that edits ACLs on existing znodes, transferring permissions from `world` to an admin user of our choice.

```
JAAS_CONFIG=$KAFKA_HOME/config/kafka_server_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/zookeeper-security-migration.sh \
    --zookeeper.connect localhost:2181 \
    --zookeeper.acl secure
```

This operation will take a few seconds and should exit without outputting anything to the console.

The migration step is only necessary for existing ZooKeeper ensembles that have previously been initialised by Kafka. Fresh ZooKeeper installations (with no prior data) will function securely without migration, provided that SASL has been enabled prior to connecting Kafka brokers.

Configure the broker to authenticate to ZooKeeper

The next step is to enable ZooKeeper authentication on the broker. Edit `server.properties`, adding the following line:

```
zookeeper.set.acl=true
```

Then, start the broker with the client-side JAAS configuration created in the previous step:

```
JAAS_CONFIG=$KAFKA_HOME/config/kafka_server_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/kafka-server-start.sh \
    $KAFKA_HOME/config/server.properties
```

Enable ZooKeeper authentication on CLI tools

The reader may have noticed a pattern among the applications that connect to a SASL-enabled ZooKeeper: they all need to set `java.security.auth.login.config` to the location of a JAAS file.

CLI tools are no exception. Once ZooKeeper authentication has been enabled, you must supply a valid JAAS file every time you need to query or modify ZooKeeper's state. The example below shows how the `kafka-config.sh` tool can be parametrised with the location of the JAAS file. Most scripts in `$KAFKA_HOME/bin` support the use of the `KAFKA_OPTS` environment variable for passing arbitrary system properties to the process in question.

```
JAAS_CONFIG=$KAFKA_HOME/config/kafka_server_jaas.conf \
KAFKA_OPTS=-Djava.security.auth.login.config=$JAAS_CONFIG \
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --describe --entity-type users
```

The JAAS file may be omitted in the scenario where you might be browsing top-level znodes that are world-readable; for example, using the `zookeeper-shell.sh` tool.

Reverting the configuration

If ZooKeeper authentication ceases to be a requirement, it can be reverted to its original (unauthenticated) state by running `zookeeper-security-migration.sh` with the `--zookeeper.acl unsecure` flag. Afterwards, you may edit `zookeeper.properties` to remove the `authProvider.1` and `jaasLoginRenew` properties. Likewise, the `server.properties` configuration will also need its `zookeeper.set.acl` setting removed or set to `false`.

Note, the rest of the chapter and the book assumes that ZooKeeper authentication is disabled. If you have enabled it out of curiosity, now is a good time to revert the configuration.

Configuring CLI tools

Once Kafka has been configured with SSL and authentication, it isn't just the producer and consumer clients that must be configured; the built-in CLI tools and other administrative applications must also be configured with the location of the client truststore file and the parameters of the authentication flow.

Start by copying the `client.truststore.jks` file (that was generated as part of setting up SSL) to a location where a client can easily access it:

```
cp /tmp/kafka-ssl/client.truststore.jks $KAFKA_HOME/config
```

Next, create a `client.properties` file `$KAFKA_HOME/config`. The contents of the file are listed below. On the author's machine, the truststore is located in `/Users/me/opt/kafka_2.13-2.4.0/config`. You will need to replace this with the absolute path to the truststore file, as most applications don't work well with relative paths. We are going to use the `admin` user for authentication, which is fitting for administrative tools.

```
security.protocol=SASL_SSL
ssl.endpoint.identification.algorithm=https
ssl.truststore.location= \
    /Users/me/opt/kafka_2.13-2.4.0/config/client.truststore.jks
ssl.truststore.password=secret
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule \
        required \
        username="admin" \
        password="admin-secret";
```

In production deployments you are more likely to have multiple admin-like users with varying permissions. A discussion on user permissions and the broader topic of authorization is deferred until the next section.

Next, run one of the existing CLI tools to verify the configuration. In the example below, we are using `kafka-topics.sh` to list the current topics. The location of the `client.properties` file is supplied via the `--command-config` flag. Note: we are connecting to port `9094` in this example, not `9092`.

```
$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9094 \
    --command-config $KAFKA_HOME/config/client.properties --list
```

The resulting output lists the topics.

```
__consumer_offsets
getting-started
```

Configuring Kafdrop

Having configured the CLI tools, the next step is to ensure that we can connect to a secured cluster using Kafdrop. Like the CLI example, Kafdrop requires a truststore file, as well as a set of properties for configuring SSL and SASL. Unlike the CLI, Kafdrop does not require an absolute path to the truststore file; instead, you can simply leave the truststore file in Kafdrop's root directory.

Start by copying `client.truststore.jks` to the Kafdrop directory, renaming the file to `kafka.truststore.jks` in the process. This is the standard truststore filename that Kafdrop expects by default.

```
cp /tmp/kafka-ssl/client.truststore.jks \
    ~/code/kafdrop/kafka.truststore.jks
```

In addition, create a `kafka.properties` file in Kafdrop's root directory, containing the following:

```
security.protocol=SASL_SSL
ssl.endpoint.identification.algorithm=https
ssl.truststore.password=secret
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config= \
    org.apache.kafka.common.security.scram.ScramLoginModule \
        required \
        username="admin" \
        password="admin-secret";
```

The contents of `kafka.properties` are strikingly similar to the `client.properties` file used in the previous example. There is only one difference: the location of the truststore file is left unspecified. Kafdrop will default to `kafka.truststore.jks` by convention.

Next, start Kafdrop, this time connecting to `localhost:9094`:

```
java -jar target/kafdrop-3.22.0-SNAPSHOT.jar \
    --kafka.brokerConnect=localhost:9094
```

Once connected, Kafdrop will behave identically to previous examples. The only difference now is that it will connect over SSL and will authenticate itself using the `admin` user. This is important for our upcoming examples, which use Kafka's authorization capabilities. Unless Kafdrop is set up to use SASL and SSL, it will be of no use once the Kafka cluster has been fully hardened. The same can be said about built-in CLI tools and any third-party tools one might be using.

Authorization

With SSL and authentication ticked off, we are two-thirds of the way there. Authentication is necessary to identify the user principal, and that is immensely useful for allowing or blocking access to individual clients. Quite often, especially when the cluster is shared across multiple applications, the binary allow-or-deny option is insufficient; we need more fine-grained control over the actions that an authenticated client should be allowed to perform. This is where *authorization* comes into the picture.

Kafka implements authorization by way of resource-centric ACLs. An ACL specifies the following:

- **Principal** — the user performing the operation.
- **Operation** — the action being performed. For example, `Read` or `Write`.
- **Host** — an optional restriction on the addresses that the rule applies to.
- **Resource type** — the type of subject on which the action is to be performed. For example, `Topic` or `Group`.
- **Resource pattern** — a simple rule for matching the subject by its name. Rules specify a single resource by its literal name, or can group multiple resources using prefixes or wildcards.
- **Outcome** — whether the action should be allowed or denied.

Both Kafka and ZooKeeper utilise ACLs for authorization; however, their implementations are distinct and should not be confused. The table below outlines the differences.

ZooKeeper ACLs	Kafka ACLs
Apply to discrete resources.	Can specify multiple resources using prefixes and wildcards.
Restrict operations to users.	Restrict operations to users or network addresses.
Can only allow an operation.	Can allow or deny operations.

There are several supported operations:

- `Read` — view a data-centric resource (for example, a topic), but not write to its contents.
- `Write` — modify the contents of a data-centric resource.
- `Create` — create a new resource.
- `Delete` — delete a resource.
- `Alter` — change the contents of a non-data resource (for example, ACLs).
- `Describe` — get information about a resource.
- `ClusterAction` — operate on the cluster. For example, fetch metadata or initiate a controlled shutdown of a broker.
- `DescribeConfigs` — retrieve the configuration of a resource.
- `AlterConfigs` — alter the configuration of a resource.
- `IdempotentWrite` — perform an idempotent write. Idempotent writes are described in Chapter 10: Client Configuration.
- `All` — all of the above.

The operations act on resource types. The following is a list of supported resource types and related operations:

- `Cluster` — the entire Kafka cluster. Supports: `Alter`, `AlterConfigs`, `ClusterAction`, `Create`, `Describe` and `DescribeConfig`.
- `DelegationToken` — a delegation token. This resource type is different from the others in that it is bound by special rules, where permissions apply to the owner and renewer. Supports: `Describe`.
- `Group` — a consumer group. Supports: `Delete`, `Describe` and `Read`.
- `Topic` — a topic. Supports: `Alter`, `AlterConfigs`, `Create`, `Delete`, `Describe`, `DescribeConfigs`, `Read` and `Write`.
- `TransactionalId` — transactional sessions of the same logical producer. Supports: `Describe` and `Write`.

The lists above may appear confusing, especially the combination of resource types and operations. Kafka's official documentation provides a more detailed matrix of the relationships between low-level protocol messages (or API calls), operations and resource types — not that it makes things a whole lot clearer. (Not unless you are intimately familiar with the underlying protocol messages.) This might not sound very encouraging, but most ACL-related issues are diagnosed and resolved through experimentation and consulting people who have seen a similar problem before. Admittedly, it's not great, but it's the best answer there is.

To make this process less ambiguous, Kafka clients will throw specific subclasses of an `AuthorizationException` if the evaluation of rules results in a *deny* outcome for one of the five resources types. For example, a deny against a topic will result in a `TopicAuthorizationException`, a deny against a group will lead to a `GroupAuthorizationException`, and so forth.

There will be times where a developer will swear they have enabled an operation on a resource, such as a topic, only to get some obscure `ClusterAuthorizationException` at runtime. It might seem nonsensical at the time — one is trying to publish a record, but is inexplicably being knocked back at the cluster level. Confusions such as these typically occur when a client action entails multiple API calls, and one of those calls is being denied.

Enable authorization on the broker

To enable authorization, edit `server.properties`, adding the following lines:

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
super.users=User:admin
```

Restart the broker for changes to take effect.

The `super.users` property specifies a semicolon-delimited list of privileged users that are able to perform *any* operation. In this example, we specified `admin` for convenience, given that we already have this user configured, and we are also signing on with the same user for interbroker authentication. You may specify other users, but make sure the interbroker user remains in the list.

Kafka delimits usernames with semicolons because a comma is a valid character in a username that has been derived from the attributes of an SSL certificate.

As it is generally accepted, operating a secure cluster implies that both client and interbroker operations are subject to authentication and authorization controls. The interbroker operations are split into two broad classes: *cluster* and *topic*. Cluster operations refer to the actions necessary for the management of the cluster, such as updating broker and partition metadata, changing the leader and the in-sync replicas of a partition, and initiating a controlled shutdown. One of the defining responsibilities of a broker is the replication of partition data. Because of the way replication works internally, it is also necessary to grant topic access to brokers. Specifically, replicas must be authorized for both `Read` and `Describe` operations on the topics being replicated. The `Describe` permission is granted implicitly, as long as the `Read` permission is present.

> Kafka takes a *default-deny* stance, which is otherwise known as a *positive* or *additive* security model. Unless an attempted action is explicitly allowed, it is assumed to be denied. This behaviour can be inverted by adding the following value to `server.properties`:
>
> ```
> allow.everyone.if.no.acl.found=true
> ```
>
> Generally speaking, the default-deny model is recommended over the default-allow, as it takes a more conservative approach to dispensing access. One should think of it as an additive approach: starting from zero — a blank slate, adding permissions on a needs basis. At any point, it can be clearly reasoned as to who has access to a resource — by simply looking over the permissions list. Unless an *allow* rule explicitly stating the *principal* and the *resource* in question is present, then there can be no access-granting relationship between the pair of entities.

Once the broker restarts, we can use the `kafka-acls.sh` tool to view and manipulate ACLs. The command below will list all configured ACLs, internally utilising the Kafka Admin API.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --list
```

It should come up with an empty list of rules. This is to be expected as we haven't yet configured any.

When specifying a bootstrap list via the `--bootstrap-server` flag, it is also necessary to state the location of the JAAS file used for authenticating to Kafka. When Kafka is running with authorization enabled, users cannot just view and alter ACLs — not unless they have been explicitly authorized to do so. By connecting with the `admin` user, we are effectively operating in 'god mode'.

> An alternate way of viewing and setting ACLs is to connect to ZooKeeper directly. This will work provided ZooKeeper is reachable from the network where the command is run, and ZooKeeper is not running with authentication and ACLs of its own. The command below shows how `kafka-acls.sh` can be used directly with an unauthenticated ZooKeeper node.
>
> ```
> $KAFKA_HOME/bin/kafka-acls.sh \
>     --authorizer-properties zookeeper.connect=localhost:2181 \
> ```

```
        --list
```

At this point, we will have a functioning broker with no ACLs. This means that, with the exception of the `admin` user, no other client will be able to interact with Kafka. Of course, the point of ACLs is not to lock out all users, but restrict access to those users who legitimately need to view or manipulate specific resources in the cluster. Previously, in the SASL examples, the user `alice` published to the `getting-started` topic. Running this example now results in a series of warnings and errors:

```
21:55:56/1854  WARN  [kafka-producer-network-thread | producer-1]: 
    [Producer clientId=producer-1] Error while fetching metadata 
    with correlation id 4 : 
    {getting-started=TOPIC_AUTHORIZATION_FAILED}
21:55:56/1854  ERROR [kafka-producer-network-thread | producer-1]: 
    [Producer clientId=producer-1] Topic authorization failed for 
    topics [getting-started]
Published with metadata: null, error: org.apache.kafka.common. 
    errors.TopicAuthorizationException: Not authorized to access 
    topics: [getting-started]
```

This particular problem can be fixed by adding read and write permissions to user `alice`:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:alice \
    --operation Read --operation Write --topic getting-started
```

Which results in the following output:

```
Adding ACLs for resource `ResourcePattern(resourceType=TOPIC, 
      name=getting-started, patternType=LITERAL)`:
  (principal=User:alice, host=*, operation=WRITE, 
      permissionType=ALLOW)
  (principal=User:alice, host=*, operation=READ, 
      permissionType=ALLOW)

Current ACLs for resource `Topic:LITERAL:getting-started`:
  User:alice has Allow permission for operations: 
      Read from hosts: *
  User:alice has Allow permission for operations: 
      Write from hosts: *
```

However, this is not sufficient. As it was previously stated, some operations require several API calls. In our producer example, we have set the property `enable.idempotence` to `true` Enabling idempotence ensures that the producer maintains strict order and does not write duplicates in the event of an intermittent network error. However, this property requires cluster-level permission for the `IdempotentWrite` operation:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:alice \
    --operation IdempotentWrite --cluster
```

Resulting in:

```
Adding ACLs for resource `ResourcePattern(resourceType=CLUSTER, □
      name=kafka-cluster, patternType=LITERAL)`:
  (principal=User:alice, host=*, operation=IDEMPOTENT_WRITE, □
      permissionType=ALLOW)

Current ACLs for resource `Cluster:LITERAL:kafka-cluster`:
  User:alice has Allow permission for operations: □
      IdempotentWrite from hosts: *
```

Running the `kafka-acls.sh` command with the `--list` switch now results in a complete ACL listing, showing `alice` and her associated permissions:

```
Current ACLs for resource `Cluster:LITERAL:kafka-cluster`:
  User:alice has Allow permission for operations: □
      IdempotentWrite from hosts: *

Current ACLs for resource `Topic:LITERAL:getting-started`:
  User:alice has Allow permission for operations: □
      Read from hosts: *
  User:alice has Allow permission for operations: □
      Write from hosts: *
```

Try publishing now. It should work like a charm.

Now, let's switch back to the consumer sample that we ran as part of setting up SASL. That example authenticated as `alice` and read from the `getting-started` topic using the `basic-consumer-sample` consumer group. Running this example now results in the following error:

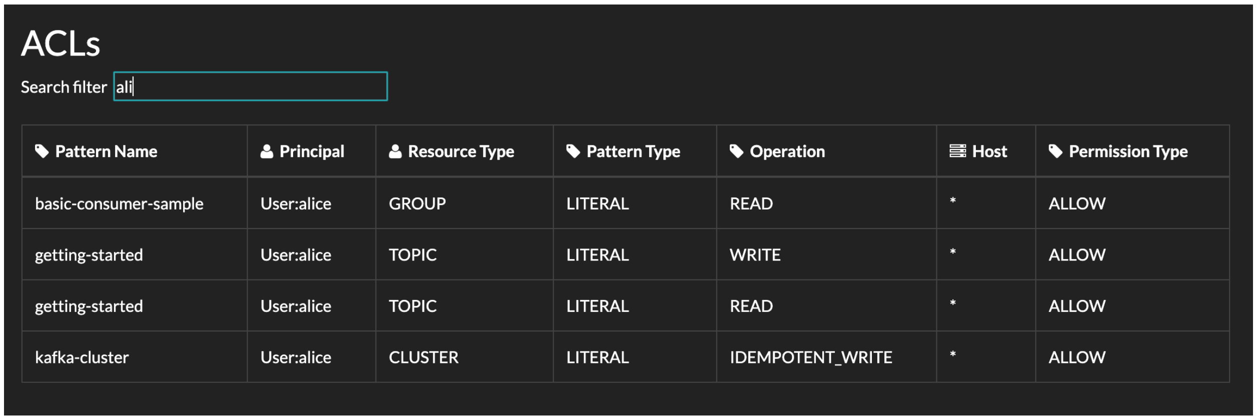
```
Exception in thread "main" org.apache.kafka.common.errors. 
    GroupAuthorizationException: Not authorized to access 
    group: basic-consumer-sample
```

The issue is that the ACLs for topics are distinct to the ACLs for consumer groups, as the two are treated as distinct resources. To fix the issue above, add the `Read` permission on the consumer group:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:alice \
    --operation Read --group basic-consumer-sample
```

The consumer should now be able to function normally. And there we have it: the world-readable `getting-started` has just been converted into a *high-assurance* topic.

ACLs can be viewed broadly for all users and resources using the Kafdrop tool. It also provides a convenient filter text box, letting you search for specific principals or resource patterns. The screenshot below shows how the ACLs we have defined so far appear in Kafdrop.


ACLs

Search filter: ali

Pattern Name	Principal	Resource Type	Pattern Type	Operation	Host	Permission Type
basic-consumer-sample	User:alice	GROUP	LITERAL	READ	*	ALLOW
getting-started	User:alice	TOPIC	LITERAL	WRITE	*	ALLOW
getting-started	User:alice	TOPIC	LITERAL	READ	*	ALLOW
kafka-cluster	User:alice	CLUSTER	LITERAL	IDEMPOTENT_WRITE	*	ALLOW

Kafdrop — ACls for the user 'alice'

In our examples, we have used `alice` to demonstrate both read (consume) and write (publish) privileges. As a convenience, this was sufficient. However, in practice, it is rare to see both the producer and the consumer acting as one entity for the same topic. More often, they are distinct. It is considered best-practice to assign individual usernames to application entities. Furthermore, an entity should only be granted the minimal set of privileges that it legitimately requires to fulfil its responsibilities.

In the same vein, the correct approach with respect to admin users is to avoid them. It is best to create a dedicated user for interbroker communications, having the necessary level of cluster and topic access, but no ability to create new users or modify ACLs.

Removing permissions

To remove a permission, simply run `kafka-acls.sh` with the `--remove` switch in place of `--add`, with the rest of the flags used for adding a permission, as shown in the example below.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --authorizer-properties zookeeper.connect=localhost:2181 \
    --remove --allow-principal User:alice \
    --operation Read --operation Write --topic getting-started \
    --force
```

The use of the `--force` switch prevents the operator from having to respond to a *yes/no* safety prompt.

Later in the chapter, we will look at bulk-removing permissions.

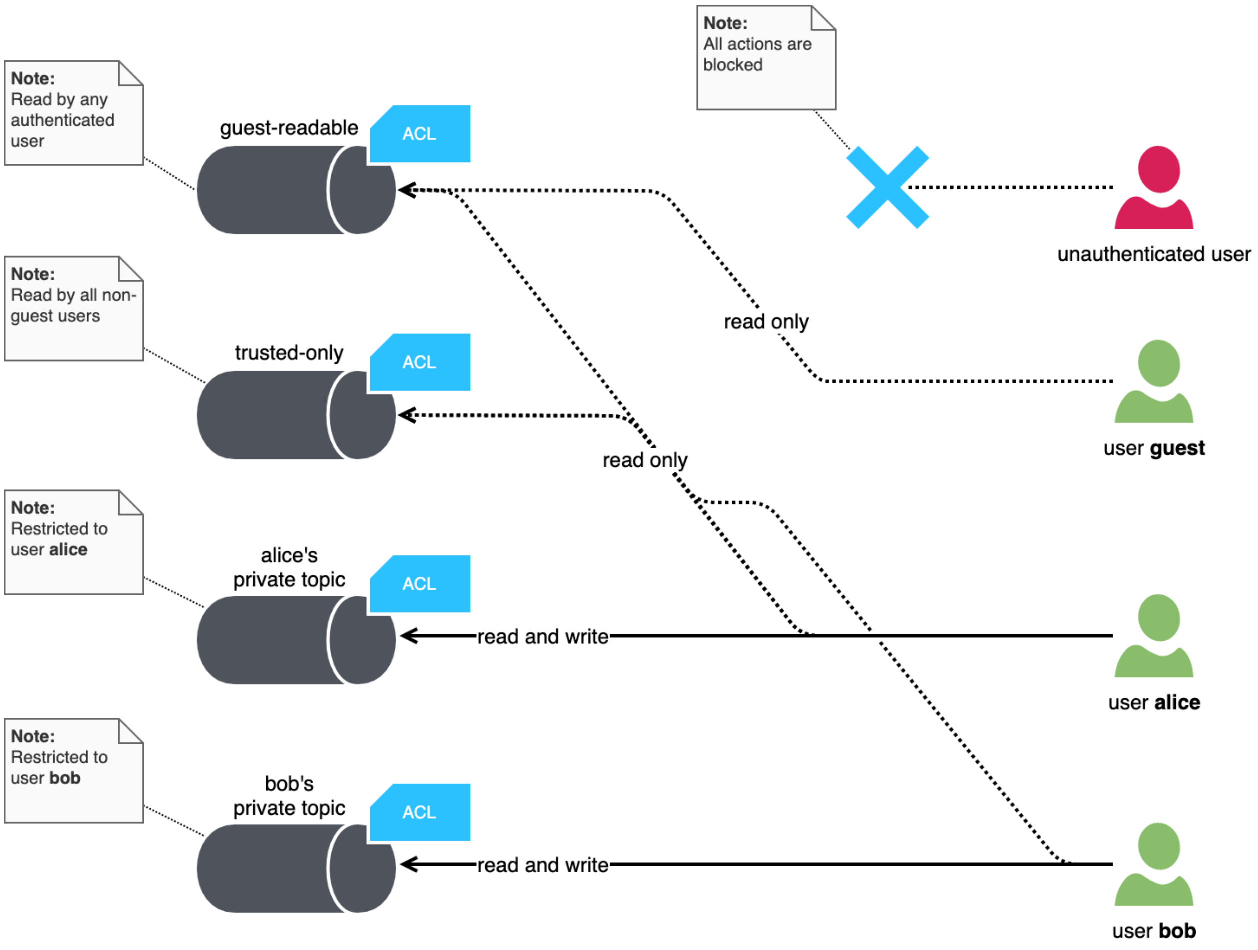
Mixing allow and deny permissions

One of the known complications of Kafka's rules is that they support both allow and deny actions. This makes them more flexible, but also easier to misconfigure. For example, you can allow access to all users on a specific topic, but then disallow access to a smaller group of users. A natural response might be: *Doesn't this contradict the default-deny model, and if not, where would this capability be useful?*

One way of looking at it, is that it allows us to interleave default-allow and default-deny tactics in the same overall security model, creating a hierarchy of sorts. To understand where this might be beneficial, consider the following challenge, involving a hypothetical Kafka cluster that permits guest level access for a small subset of resources. Guest credentials can be procured relatively easily — there might even be a pre-canned set of shared credentials that are freely available. Assuming strong network-level controls are in place, guest access is only permitted from a trusted network.

We might be publishing something trivial on a topic which we would like to make available to guests for reading, as well as to trusted users. Let's call this *low-assurance* topic `guest-readable`. Now, assume there is another topic which contains information for authenticated users that a have higher level of clearance than `guest` — a *medium assurance* topic. We will call it `trusted-only`. And finally, the cluster may contain numerous *high-assurance* topics that are admissible to specific users. Frankly, the high assurance topics are least interesting, as they can be trivially fortified using standard ACLs. They were only added to the challenge for completeness, to make it more life-like.

To make this challenge more interesting, we do not know in advance who all the users are. All that can be stated with certainty is the username of the guest user — `guest`. Any other user that is authenticated, but not `guest`, is a trusted user and should have read access to the `trusted-only` topic. The diagram below illustrates the intended target state for this challenge, depicted the relationships between users and topics.



The ACL challenge

Normally, we would add *allow* rules to the topics, until all access requirements are satisfied. The complication here is that we don't have a definitive list of usernames to work with; furthermore, even if we did, the addition of new users would require updates to a vast number of ACLs, making their maintenance unsustainable. Ideally, this problem is solved with Role-Based Access Control (RBAC); however, the out-of-the-box Kafka setup does not support this.

There are third-party extensions to Kafka that support RBAC. This book is predominantly focused on the foundational insights and advanced skills — featuring a vanilla Kafka setup, as well as the minimally essential set of third-party tools. RBAC is on the far-right of the notional 'extensibility' spectrum, and has not been considered for inclusion into the base Kafka offering. As such, we will not explore it further.

Before solving this challenge, we need to create our fixtures. We need a pair of topics — `guest-readable` and `trusted-only`, and we need the `guest` user. Let's get started by creating these entities:

```
# add a user (requires ZooKeeper access)
$KAFKA_HOME/bin/kafka-configs.sh --zookeeper localhost:2181 \
    --alter \
    --add-config 'SCRAM-SHA-512=[password=guest-secret]' \
    --entity-type users --entity-name guest


# create the guest-readable topic
$KAFKA_HOME/bin/kafka-topics.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --create --topic guest-readable \
    --partitions 1 --replication-factor 1


# create the trusted-only topic
$KAFKA_HOME/bin/kafka-topics.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --create --topic trusted-only \
    --partitions 1 --replication-factor 1
```

The `kafka-topics.sh` CLI supports the creation of topics via a bootstrap list, provided you give it the appropriate credentials and truststore. On the other hand, the `kafka-configs.sh` tool is limited in the entity types that can be configured via a broker. Entities such as users must be configured directly in ZooKeeper.

The next step is to try reading from the `guest-readable` topic using the `guest` user. Rather than using our existing SASL consumer example, we will change the code slightly to run it using a free consumer — in other words, without an encompassing consumer group. This particular variation replaces the `Consumer.subscribe()` call with `Consumer.assign()`, removing the need to pass in the `group.id` consumer property. The complete listing is shown below.

```
import static java.lang.System.*;

import java.time.*;
import java.util.*;
import java.util.stream.*;

import org.apache.kafka.clients.*;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.*;
import org.apache.kafka.common.config.*;
import org.apache.kafka.common.security.scram.*;
import org.apache.kafka.common.serialization.*;

public final class SaslSslFreeConsumerSample {
  public static void main(String[] args) {
    final var topic = "guest-readable";

    final var loginModuleClass = ScramLoginModule.class.getName();
    final var saslJaasConfig = loginModuleClass
        + " required\n"
        + "username=\"guest\"\n"
        + "password=\"guest-secret\";";

    final var config = new HashMap<String, Object>();
    config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9094");
    config.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
               "SASL_SSL");
    config
    .put(SslConfigs.SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG,
         "https");
    config.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,
               "client.truststore.jks");
    config.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG, "secret");
    config.put(SaslConfigs.SASL_MECHANISM, "SCRAM-SHA-512");
    config.put(SaslConfigs.SASL_JAAS_CONFIG, saslJaasConfig);
    config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName());
    config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName());
    config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
    config.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
    try (var consumer = new KafkaConsumer<String, String>(config)) {
      final var partitionInfos = consumer.partitionsFor(topic);
      final var topicPartitions = partitionInfos.stream()
          .map(partInfo -> new TopicPartition(partInfo.topic(),
                                              partInfo.partition()))
          .collect(Collectors.toSet());
      consumer.assign(topicPartitions);

      while (true) {
        final var records = consumer.poll(Duration.ofMillis(100));
        for (var record : records) {
          out.format("Got record with value %s%n", record.value());
        }
      }
    }
  }
}
```

Running this code will produce the following error:

```
Exception in thread "main" org.apache.kafka.common.errors. ↩
    TopicAuthorizationException: Not authorized to access ↩
    topics: [guest-readable]
```

And indeed, that is to be expected — after all, we had just created the `guest` user. We need to grant the due permissions to the `guest` user. But here's the snag: if we keep granting permissions additively, then we have to specify all users. And as it was stated in the challenge, the usernames are now known in advance.

Rather than allowing `guest-readable` for all known users individually, we simply make it world-readable. Here 'world' is a bit of a misnomer — we aren't actually exposing the topic to the world — only to those users that have successfully authenticated via the `SASL_SSL` listener. Run the command below.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:"*" \
    --operation Read --topic guest-readable
```

Compared to the earlier examples, we are using a wildcard (quoted asterisk character) in place of a username. The wildcard can be specified using single or double-quotes. Avoid using the wildcard without quotes, as most shells will expand the `*` character to a list of files in the current directory of the caller.

To make the testing easier, we should also publish some records to the `guest-readable` and `trusted-only` topics. This can be accomplished using the `kafka-console-producer.sh` CLI:

```
$KAFKA_HOME/bin/kafka-console-producer.sh \
    --producer.config $KAFKA_HOME/config/client.properties \
    --broker-list localhost:9094 \
    --topic guest-readable
```

Key in a few records, and press `CTRL-D` when done. Repeat for the `trusted-only` topic.

Run our consumer code again. This time it should work. You can also change the user from `guest` to `alice`, and it will still work as expected.

With the low-assurance topic out of the way, the next part of the challenge is to make our medium-assurance `trusted-only` topic behave as intended. The solution, as the reader would have guessed, is to mix allow and deny outcomes.

Start by making the `trusted-only` topic world-readable:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:"*" \
    --operation Read --topic trusted-only
```

Then, follow up by excluding the `guest` user from the `trusted-only` topic, using the `--deny-principal` flag:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --deny-principal User:guest \
    --operation Read --topic trusted-only
```

Essentially, the above is saying: *"make 'trusted-only' readable by everyone except the 'guest' user"*, which is exactly what is needed.

Run the code again, switching the user to `guest` and the topic to `trusted-only`. We should see a `TopicAuthorizationException`. However, if we switch the user to `alice`, the topic's contents will be revealed.

Deny rules take precedence over *allow* rules, irrespective of the granularity of the resource patterns identified by those rules. Therefore, you should always use an *allow* rule over a broader-matching resource pattern than a *deny* rule. Stated otherwise, deny a subset of the resources that you had previously allowed, not the other way round. Had we attempted to reverse the rules of the challenge — deny everyone but allow a specific user, it would not have worked — the *deny* rule would have had the final say. In practice, when combining *allow* and *deny* outcomes, the outer-most *deny* rule is implicit — enforced by Kafka's own *default-deny* policy. This gives us the flexibility of (up to) three rule tiers — the outer-most *default-deny*, followed by a custom *allow* rule, and finally by a custom *deny* rule.

Literal and prefixed resource patterns

Most ACL examples so far have involved the literal matching of resources and user principals. In other words, we specified the *exact* name of the topic and the *exact* username, along with a concrete allow/deny outcome. These are called *literal* matches. In one example, when setting up world-readable topics, we resorted to the use of the `User:"*"` *wildcard* match on the user principal.

It should be noted that wildcards in Kafka ACLs behave very differently to the familiar wildcards used to match files and directories in the UNIX and Windows command shells. When used on its own, a wildcard implies *"match all entities"*. However, a wildcard cannot be combined with literal text to mean *"match a part of a string"*.

Kafka resolved the issue of partial resource matching in KIP-290[41] as part of release 2.0.0 — with the introduction of prefixed ACLs, addressing some of the long-standing limitations of its multitenancy capabilities.

The default pattern type is *literal*, which is the equivalent of invoking `kafka-acls.sh` with the `--resource-pattern-type=literal` flag. Partial matches can be accomplished using the `--resource-pattern-type=p`
For example, the following denies `guest` from all topics that begin with the literal string `trusted`:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --deny-principal User:guest \
    --resource-pattern-type=prefixed \
    --operation Read --topic trusted
```

Literal strings and wildcards can be used to identify resources, as well as to identify user principals. Prefixed matching applies only to resources — it is not possible to prefix-match a user.

[41]https://cwiki.apache.org/confluence/x/QpvLB

Listing and bulk-removal of permissions using the CLI

It has been shown how rules can be created and removed using the CLI. The most convenient way to list rules, provided you are working in a browser environment, is via Kafdrop's ACL screen. In those cases where Kafdrop is not at hand, the `kafka-acls.sh` can be used to query rules and even remove them in bulk.

We've already seen how the `--list` switch can be used to display the rules. But a production environment containing hundreds or thousands of ACL entries makes the unfiltered `--list` command unwieldy. Running `--list` (with no other predicates) on the state of the ACLs accumulated from all previous examples results in the following:

```
Current ACLs for resource `Topic:LITERAL:guest-readable`:
  User:* has Allow permission for operations: Read from hosts: *

Current ACLs for resource `Cluster:LITERAL:kafka-cluster`:
  User:alice has Allow permission for operations: ⏎
      IdempotentWrite from hosts: *

Current ACLs for resource `Topic:PREFIXED:trusted`:
  User:guest has Deny permission for operations: Read from hosts: *

Current ACLs for resource `Topic:LITERAL:getting-started`:
  User:alice has Allow permission for operations: Read from hosts: *
  User:alice has Allow permission for operations: Write from hosts: *

Current ACLs for resource `Topic:LITERAL:trusted-only`:
  User:guest has Deny permission for operations: Read from hosts: *
  User:* has Allow permission for operations: Read from hosts: *

Current ACLs for resource `Group:LITERAL:basic-consumer-sample`:
  User:alice has Allow permission for operations: Read from hosts: *
```

The output is large but still useful. In fact, we can observe that the `Deny` permission for the literal `trusted-only` topic is redundant, as it is covered by the broader-matching `Deny` permission on the prefix `trusted` topic; both targeting the same user principal `alice`.

The basic `--list` switch can be elaborated upon using two additional resource pattern types: `any` and `match`. The `any` pattern type locates all rules that match the specified resource name *exactly*, which includes literal, wildcard and prefixed patterns.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --list --resource-pattern-type=any \
    --topic trusted-only
```

This command results in the following:

```
Current ACLs for resource `ResourcePattern(resourceType=TOPIC, □
    name=trusted-only, patternType=LITERAL)`:
  (principal=User:*, host=*, operation=READ, permissionType=ALLOW)
  (principal=User:guest, host=*, operation=READ, permissionType=DENY)
```

The `any` pattern type has located two rules of the three that apply to the `trusted-only` topic. It didn't match the prefixed pattern for the name `trusted`, even though the latter also applies to the `trusted-only` topic. To include all rules that affect a given resource, change the pattern type type to `match`. The result:

```
Current ACLs for resource `ResourcePattern(resourceType=TOPIC, □
    name=trusted-only, patternType=LITERAL)`:
  (principal=User:*, host=*, operation=READ, permissionType=ALLOW)
  (principal=User:guest, host=*, operation=READ, permissionType=DENY)

Current ACLs for resource `ResourcePattern(resourceType=TOPIC, □
    name=trusted, patternType=PREFIXED)`:
  (principal=User:guest, host=*, operation=READ, permissionType=DENY)
```

Splendid. All three rules have been located. The `match` pattern type is convenient for debugging ACLs issues and reasoning about the access rights to a specific resource. Using a `match` query effectively lets us take a look at ACLs with the eyes of Kafka's built-in `SimpleAclAuthorizer`.

The ability to round up several rules for a resource is also convenient when performing a bulk delete operation. To remote the matching rules, run the commands above using the `--remove` switch in place of `--list`. Bear in mind that the remove command does not preview the rules before deleting them, although it does provide for a *yes/no* safety prompt, unless executed with the `--force` switch. Consequently, always run the `--list` command to ascertain the complete list of affected rules before removing them.

In the examples above, we first created the `guest` user, then the two topics, then associated the user with the topics through several ACL rules. Kafka allows us to carry out these steps in any order. When creating a rule, there is no requirement that either the principal or the resource must exist. There is no referential dependency relationship between these three entities. As a consequence, it is also possible to delete users and topics despite having ACL rules that reference them.

As much as it is a convenience, it is also a 'gotcha': it is possible to accidentally create a rule that references a mistyped user or topic — provided the rule is well-formed, Kafka will vacuously persist it. When creating ACL rules, particularly when these rules relate to high-assurance resources, it is considered best-practice to validate these rules. The best way to do this is programmatically, by writing a script that performs specific actions and asserts the allow or deny outcomes. However, the cost of getting the test wrong may be catastrophic — imagine asserting a `Delete` operation that succeeds when it is expected to fail. The result is the deletion of resources — not just a failure of the test, but the loss of potentially vital data and the disruption of downstream consumers.

The recommended way to test rules is to operate two clusters — a *staging* and a *production* setup — where the production ACLs are only manipulated via pre-canned scripts that have been validated against the staging cluster. This strategy implies that users are mirrored in both clusters and have identical permissions. In that sense, a staging cluster is quite different from a typical non-production cluster, where users might routinely be given elevated permissions to enable productivity.

Network address restrictions

In addition to restrictions on user principals, Kafka allows rules to specify the network addresses of connected clients. This is done with the `--allow-host` or `--deny-host` flags, with their sole argument being the target IP address. Multiple `--allow-host` and `--deny-host` flags may be specified in a single command, resulting in the addition of multiple rules.

The hosts passed to `--allow-host` and `--deny-host` must be specific addresses — in either IPv4 or IPv6 form. Hostnames and IP address ranges are not presently supported. KIP-252[42] proposes to extend the existing ACL functionality to support address ranges as well as groups of addresses in CIDR notation; however, this KIP is still in its infancy.

While the option is present, the changing landscape of application architecture — the proliferation of Cloud-based technologies, containerisation, NAT, and elastic deployment topologies, make it difficult to recommend host-based rules as a legitimate form of application-level access control. It is becoming more challenging and sometimes downright intractable to isolate application processes based on their network address. Where this makes sense, controlling access based on origin addresses is best accomplished with a firewall — a device that is designed for this very purpose and provides effective network and transport-layer controls. Furthermore, firewalls understand ports, protocols and, crucially, address ranges. The latter represents an absolute requirement when dealing with elastic deployments. At the application layer, access should be a function of user principals and, ideally, their roles.

[42]https://cwiki.apache.org/confluence/x/jB6HB

Common authorization scenarios

Although there are numerous supported operations and resource types supported by Kafka's authorization model, the majority of authorization needs fall under a handful of use cases. These are presented here.

Creating topics

The creation of a topic requires `Create` privileges on the `Topic` resource type. The topic name can be literal, although more often topic creation and publishing rights are granted over a prefixed resource pattern. In the example below, user `bob` is granted permission to create any topic that begins with the string `prices.`.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --resource-pattern-type=prefixed \
    --operation Create --topic prices.
```

Deleting topics

Topic deletion privileges are the logical opposite of creation, replacing the `Create` operate with `Delete`:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --resource-pattern-type=prefixed \
    --operation Delete --topic prices.
```

Publishing to a topic

Like creating a topic, publishing is often granted as a prefixed permission — requiring `Write` capability on the `Topic` resource type.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --resource-pattern-type=prefixed \
    --operation Write --topic prices.
```

If the publisher requires idempotence enabled, then a further `IdempotentWrite` operation must be allowed for the `Cluster` resource type:

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --operation IdempotentWrite --cluster
```

Consuming from a topic

For a free consumer to read from a topic, it requires the `Read` operation on the `Topic` resource type. Although consumer permissions may be specified using prefixed patterns, the more common approach is to issue literal credentials to minimise unnecessary topic exposure — in line with the *Principle of Least Privilege*. To grant `bob` read rights to the `prices.USD` topic, issue the following command.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --operation Read --topic prices.USD
```

If the consumer operates within the confines of a consumer group, then the `Read` operation on the `Group` resource type is required. Often, the consumer group is named after the user because that's how the user-group relationship tends to unfold in practice. A consumer group is effectively a private load-balancer, and in the overwhelming majority of cases, the partition load is distributed across instances of the same application. So it's safe to assume that users will want to create their own consumer groups. This can be accomplished using a prefix-based convention. To grant `bob` rights to all consumer groups beginning with `bob.`, run the following command.

```
$KAFKA_HOME/bin/kafka-acls.sh \
    --command-config $KAFKA_HOME/config/client.properties \
    --bootstrap-server localhost:9094 \
    --add --allow-principal User:bob \
    --resource-pattern-type=prefixed \
    --operation Read --group bob.
```

The facets of information security are numerous and varied, and it can be said without exaggeration that securing Kafka is an epic journey. If there is one thing to be taken away from this chapter, it is that threats are not confined to external, anonymous actors. Threats can exist internally and may persist for extended periods of time, masquerading as legitimate users and occasionally posed by them.

The correct approach to securing Kafka is a layered one. Starting with network policy, the objective is to logically segregate networks and restrict access to authorized network segments, using virtual networking if necessary to securely attach edge networks to operational sites. Network policies block clients operating out of untrusted networks and erect barriers to thwart internal threads, restricting their ability to access vital infrastructure components.

Having segregated the network, the attention should be turned to assuring the confidentiality and integrity of communications. This is accomplished using transport layer security, which in Kafka is implemented under the SSL moniker. The use of X.509 certificates complements SSL, providing further assurance to the clients as to the identity of the brokers — verifying that the brokers are who they claim to be. SSL can be applied both to client-to-broker and interbroker communications. We also have learned of the present limitations in Kafka *vis à vis* SSL — namely, the inability to use SSL to secure broker-to-ZooKeeper communications.

SSL can also be used in reverse, acting as an authentication mechanism — attesting the identity of the connected client to the broker. On the point of authentication, Kafka offers a myriad of options using SASL. These range from the enterprise-focused GSSAPI (Kerberos), to PLAIN and SCRAM schemes for username/password-based authentication, OAuth 2.0 bearer tokens, and finally to delegation tokens that simplify the authentication process for large hordes of ephemeral worker nodes. Authentication isn't just limited to broker nodes; it can also be enabled on ZooKeeper — limiting one's ability to view and modify the contents of potentially sensitive znodes.

Authentication, as we have learned, is really a stepping stone towards authorization. There is little benefit in knowing the identity of a connected party if all parties are to be treated equally. Parties — being clients and peer brokers — bear different roles within the overall architecture landscape. They have different needs and are associated with varying levels of trust. Kafka combines elaborate, rule-based ACLs with a default-deny permission model to create a tailored access profile for every client.

The layered security controls supported by Kafka, in conjunction with externally-sourced controls — such as firewalls, VPNs, transparent TLS proxies and storage encryption — collectively form an environment that is impregnable against a broad range of threat actors. They enable legitimate actors to operate confidently in a secure environment.