Kong Gateway Operations for Kubernetes

Securing Services on Kong

Course Agenda

1. Kong Gateway Installation on K8s
2. Intro to Kong Ingress Controller
3. Securing Kong Gateway
4. **Securing Services on Kong**
5. OIDC Plugin
6. Kong Vitals
7. Advanced Plugins Review
8. Troubleshooting
9. Test your Knowledge

Learning Objectives

1. Understand how to secure the data plane on Kong
2. Deploy and configure the Rate limiting plugin
3. Configure Authentication with the JWT and Mutual TLS plugins
4. Identify some general Kong data plane hardening measures

Reset Environment

Before we start this lesson, be sure to reset your environment:

```
$ cd /home/labuser
$ git clone https://github.com/Kong/edu-kgac-202.git
$ source ./edu-kgac-202/base/reset-lab.sh
```

Securing API Services

There are a number of security aspects that need consideration when setting up services on Kong. These may relate to:

- Authentication
- Bot Detection
- Data encryption
- IP Restriction
- Threat detection
- DDOS
- Interception

Kong covers many of these aspects by implementing plugins.

In this section we will look at two aspects in particular - Authentication and DDOS prevention with Rate Limiting.

Rate Limiting

What is the Rate Limiting Plugin?

- Rate limiting is an important part of API security, as DoS attacks and intended/unintended overuse can overwhelm a server with unlimited API requests.
- Rate Limiting Plugin is used to limit the number of HTTP(S) requests allowed, made in a given period of time, from seconds to years, to maintain QoS and overall availability.
- Rate Limiting Plugin is a simple and effective way to protect APIs against malicious attacks and overuse and is one of Kong's most popular traffic control add-ons.
- Shared services need to protect themselves from excessive use, so rate limiting on both client and server side is crucial to avoid impact on availability and cascading failures.

Rate Limiting Use Cases

- Protection against DDoS attacks
- Protection against API abuse
- Protection against inadvertent overuse
- Protection against brute force attacks
- Limit sensitive data exposure
- Protect computationally and/or financially intensive endpoints served by auto-scaling

Rate Limiting Plugin Usage

- The Rate Limiting Plugin can limit usage based on IP address, Consumer, Service & HTTP Header, and can be scoped to Consumer, Route, Service or globally.
- This plugin is used to protect endpoints against intentional, inadvertent or overuse and DDoS / Brute Force attacks.
- Multiple instances of the Rate Limiting plugin can be applied to services, routes and/or consumers to provide fine-tuned control.

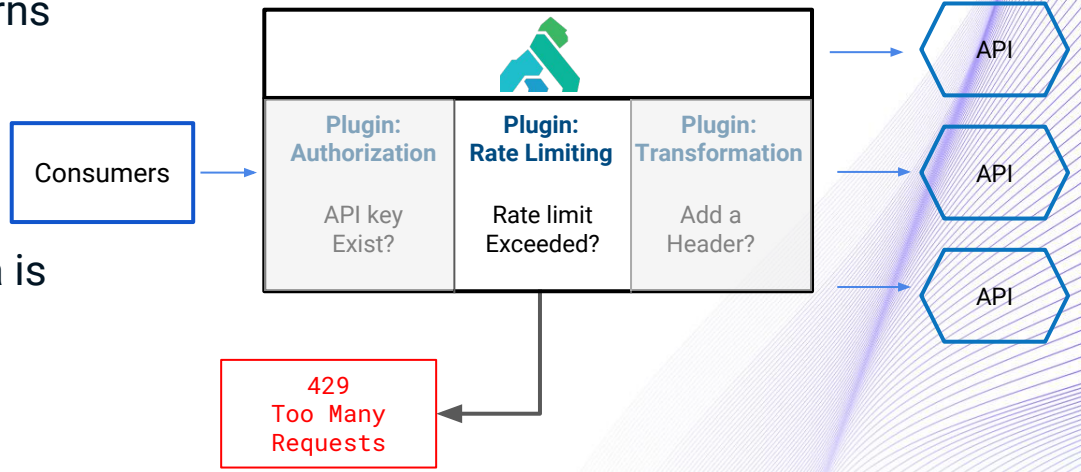
Rate Limiting Plugin Data Flow

Kong receives client's request, and based on the plugin setup and client criteria, provides the response through normal flow, or returns HTTP 429 status code with this JSON body:

```
{"message": "API rate limit exceeded"}
```

When this plugin is enabled, Kong returns headers to the client indicating

- the allowed limits
- requests remain available
- the time remaining until the quota is reset.



Enabling Rate Limiting Plugin

The Rate Limiting Plugin can be configured through Kong Manager, Admin API, Declaratively or by using **KIC CRDs**

The following example illustrates the plugin scoped to a service, with counters stored in the datastore and shared across nodes, and configured to keep proxying requests in case of loss of connectivity to the datastore. Note that this is using the Admin API method and we'll be using CRDs:

```
$ http --form POST http://testHost:30001/services/testService/plugins \
    name=rate-limiting \
    config.second=8 \
    config.min=384 \
    config.hour=32166 \
    config.year=40000000 \
    config.policy=cluster \
    config.limit_by=consumer \
    config.fault_tolerant=true
```

Counter Storage Policies

Counter storage policies control how Kong stores request counters, and can have ramifications based on the use-case.

- **‘local’:** counters stored locally in-memory on the node. This is the simplest strategy to implement with minimal performance impact. However it is less accurate, and could diverge when scaling, unless a consistent-hashing load balancer is used.
- **‘cluster’**: counters stored in Kong’s datastore and shared across nodes. Each requests forces a read/write, so relatively largest impact on performance among these policies. However, it is accurate and no extra components are needed to implement.
- **‘redis’**: counters stored on a Redis server, external to Kong. Performance impact is more than ‘local’, but less than ‘cluster’. It is accurate, but needs installation/configuration of Redis as an extra component.

Counter Storage Policies

Counter storage policies control how Kong stores request counters, and can have ramifications based on the use-case.

Mode	Strategy	Pros	Cons
Local	Counters stored locally in-memory on the node	Simplest strategy to implement with minimal performance impact.	Less accurate Could diverge when scaling, unless a consistent-hashing load balancer is used.
Cluster	Counters stored in Kong's datastore and shared across nodes	Accurate and no extra components needed to implement	Each requests forces a read/write, so relatively largest impact on performance among the three policies.
Redis	Counters stored on a Redis server, external to Kong.	It is accurate	Performance impact is more than 'local', but less than 'cluster'. Needs installation/configuration of Redis as an extra component.

Implementation Considerations

- For general back-end protection, where accuracy is of less importance, ‘local’ would work. You need to fine tune based on number of nodes in Kong cluster and load balancing policy. In cases where accuracy is important, such as financial/military, recommendation is to start with ‘cluster’ and move to ‘redis’ in case of drastic performance impact.

- In a Kong DB-less setup, this plugin is not supported with the cluster strategy, as it requires to be able to write the counters to the database. Furthermore, this plugin does not support the cluster strategy in Hybrid mode, so only ‘local’ and ‘redis’ can be used for data planes.

- Default entity used when aggregating limits is set to ‘consumer’, and could have other values such as '`header`', '`path`', '`service`', '`credential`' and '`ip`'. When selected entity can't be retrieved, due to issues such as missing header/service, the plugin will fallback to use IP as identifier.

Task: Deploy an Ingress for our httpbin app

```
$ cd $HOME/edu-kgac-202
$ kubectl apply -f ./base/httpbin-ingress.yaml
namespace/httpbin-demo created
service/httpbin-service created
ingress.networking.k8s.io/httpbin-ingress created
$ http --headers get $KONG_PROXY_URL/httpbin
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
------
$
```

Task: Configure Rate Limiting and key-auth Plugins and add Jane the Consumer

```
$ kubectl apply -f ./exercises/rate-limiting/httpbin-ingress-rates-key.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-auth created
kongplugin.configuration.konghq.com/httpbin-rate-limiting created
service/httpbin-service configured
ingress.networking.k8s.io/httpbin-ingress configured
$ kubectl apply -f ./exercises/rate-limiting/jane-consumer.yaml
namespace/httpbin-demo unchanged
secret/jane-apikey created
kongconsumer.configuration.konghq.com/jane created
$
```

Task: Create Some Traffic for User

Copy and paste the command, and leave it running while you perform the following steps. This will proxy a request every second for 20 seconds:

```
$ (for ((i=1;i<=20;i++))
     do
     sleep 1
     http --headers $KONG_PROXY_URL/httpbin?apikey=JanePassword
   done)
```

```
HTTP/1.1 429 Too Many Requests
...
RateLimit-Reset: 42
X-RateLimit-Limit-Minute: 5
Retry-After: 42
X-RateLimit-Remaining-Minute: 0
RateLimit-Limit: 4
{
```

You'll quickly see the rate limit being exceeded.

Troubleshooting Tips/Tricks

When this plugin is enabled, Kong sends additional headers back to the client with useful information on the operation of the plugin:

```
RateLimit-Limit: 5 # Allowed limits
RateLimit-Remaining: 4 # Remaining available requests
RateLimit-Reset: 58 # Remaining seconds until quota is reset
X-RateLimit-Limit-Minute: 5 # Allowed limits
X-RateLimit-Remaining-Minute: 4 # Remaining available requests
```

If more than one limit is set, a combination of time limits will be included in the response headers by Kong:

```
X-RateLimit-Limit-Second: 5
X-RateLimit-Remaining-Second: 4
X-RateLimit-Limit-Minute: 10
X-RateLimit-Remaining-Minute: 9
```

Authentication

What are Authentication Plugins?

Authentication Plugins provide a mechanism to validate that users are whom they claim to be, protecting the backend API against unauthorized calls.

There are a number of Authentication Plugins provided by Kong covering many technologies including, but not limited to:

- Okta
- Vault Authentication
- OpenID Connect
- Basic Authentication
- HMAC Authentication
- Key Authentication
- Mutual TLS Authentication
- JWT Authentication
- LDAP
- OAuth 2.0

We will take a closer look and deploy one of these.

JWT Authentication Plugin

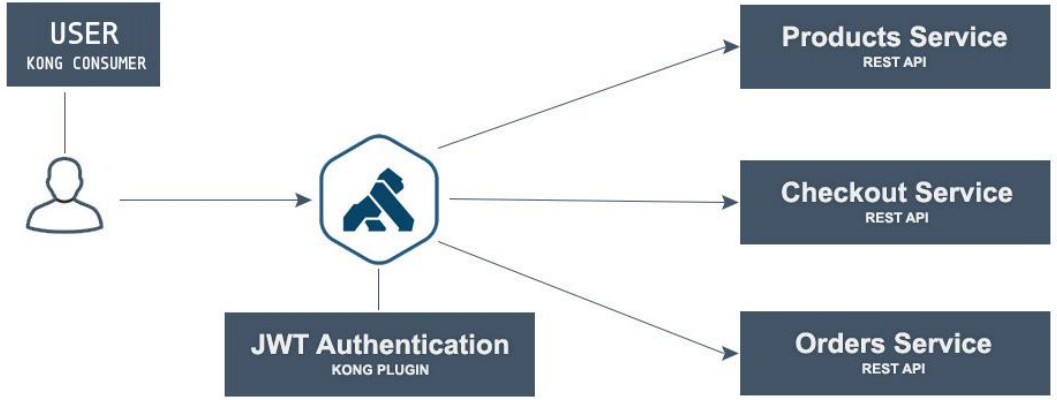
What is JWT Token?

- JSON Web Token (JWT) is an open standard that allows two parties to securely exchange information like claims as JSON objects in a compact and self-contained way.

- A JWT has a data payload signed by a trusted party to prevent spoofing, and an authorizer verifies that the JWT token is authentic, allowing (or forbidding) access to that resource. Typically, a JWT payload is not encrypted.

What is JWT Auth Plugin?

In this approach, the plugin serves as the JWT authorizer and authenticates the JWT in the HTTP request by verifying token's claims and ensuring a trusted party signed it. Then, depending on whether these steps were successful, Kong Gateway routes the upstream service request.



Each Consumer will have JWT credentials (public and secret keys), which must be used to sign their JWTs. A token can then be passed through a query string parameter, a cookie, or HTTP request headers.

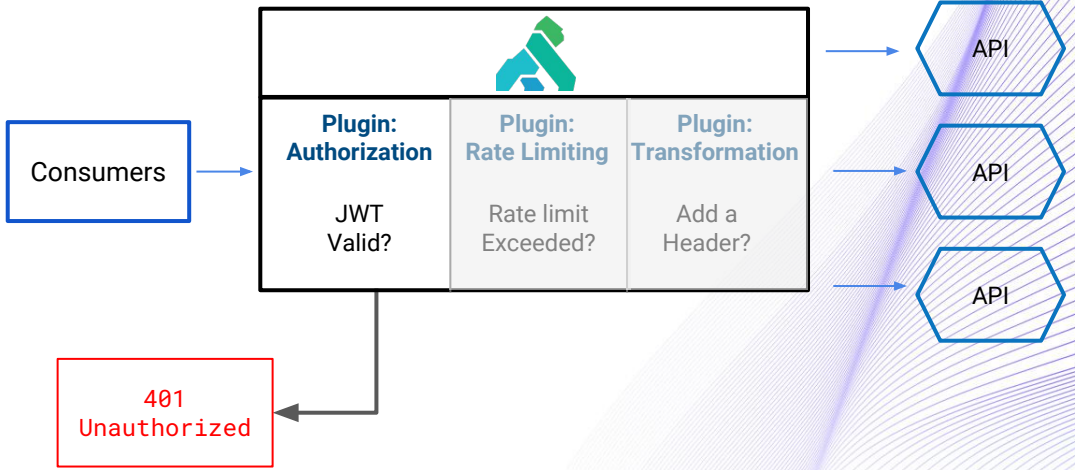
JWT Use Cases

- Securing access to upstream APIs through consumer JWT credentials as authentication mechanism.
- Issued tokens can be validated independent of the Authorization Server, e.g. using Kong API Gateway.
- JWT tokens are “stateless”, meaning that session information is not stored server-side, which saves memory. Also JWT tokens can be used to authenticate users on multiple applications, sharing same private key to verify validity.
- JWTs are a good way of securely transmitting information between parties. As JWTs are singed, for example, using public/private key pairs, and the signature is calculated using the header and payload, both sender and content can be validated.

JWT Plugin Data Flow

Kong receives client's request, with JWT supplied through a query string parameter, a cookie, or HTTP request headers.

A valid JWT proceeds through, while an invalid one results in HTTP 401 status code with a JSON message indicating bad token, invalid signature etc.



Enabling JWT Plugin

The JWT Plugin can be configured through Kong Manager, admin API or Declaratively. You can configure it using admin API, by posting to /plugins endpoint under the desired scope of global/consumer/route/service.

Here is an example of the plugin scoped to a service:

```
$ http -f POST http://testHost:30001/services/httpbin-service/plugins \
    name=jwt \
    config.secret_is_base64=false \
    config.run_on_preflight=true
```

Using JWT Plugin

To use the plugin, you need to :

1. Create a Consumer, as a representation of a developer using the final service.
2. Associate one or more JWT credentials, holding public/private keys used to verify the token, to that consumer.
3. Generate the token for the consumer, for example using JWT debugger at https://jwt.io with the header (RS256), claims (iss), and secret associated with the key.
4. Supply JWT through a query string parameter, a cookie, or HTTP request headers, when consuming the service.
5. You can list the keys for the consumer using

```
http GET http://{HOST}:30001/consumers/consumerName/jwt
```

Task: Reset httpbin service

```
$ cd ~/edu-kgac-202/exercises/jwt
$ kubectl delete ns httpbin-demo
namespace "httpbin-demo" deleted
$ kubectl apply -f ../../base/httpbin-ingress.yaml
namespace/httpbin-demo created
service/httpbin-service created
ingress.networking.k8s.io/httpbin-ingress created
$ http --headers get $KONG_PROXY_URL/httpbin
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
------
```

Task: Enable JWT Plugin for our Service

```
$ kubectl apply -f ./httpbin-ingress-jwt.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-jwt created
service/httpbin-service configured
ingress.networking.k8s.io/httpbin-ingress configured
$ http --headers get $KONG_PROXY_URL/httpbin
HTTP/1.1 401 Unauthorized
Connection: keep-alive
Content-Length: 26
Content-Type: application/json; charset=utf-8
Date: Fri, 03 Jun 2022 21:27:02 GMT
Server: nginx
X-Kong-Response-Latency: 1
$
```

Task: Create a consumer and assign JWT credentials

Create KongConsumer jane

```
$ cat << EOF > jane-consumer.yaml
apiVersion: configuration.konghq.com/v1
kind: KongConsumer
metadata:
  name: jane
  namespace: httpbin-demo
  annotations:
    kubernetes.io/ingress.class: kong
username: jane
EOF
$ kubectl apply -f ./jane-consumer.yaml
```

Task: Create a consumer and assign JWT credentials

Create key and secret for jane and stage to Kubernetes

```
$ openssl genrsa -out ./jane.pem 2048
$ openssl rsa -in ./jane.pem -outform PEM -pubout -out ./jane.pub
$ kubectl create secret generic jane-jwt -n httpbin-demo \
  --from-literal=kongCredType=jwt  \
  --from-literal=key="jane-issuer" \
  --from-literal=algorithm=RS256 \
  --from-file=rsa_public_key=./jane.pub \
  -o yaml --dry-run=client > ./jane-secret.yaml
$ kubectl apply -f ./jane-secret.yaml
```

Task: Create a consumer and assign JWT credentials

Create jane KongConsumer and add the jwt credential

```
$ cat << EOF > jane-consumer-jwt.yaml
apiVersion: configuration.konghq.com/v1
kind: KongConsumer
metadata:
  name: jane
  namespace: httpbin-demo
  annotations:
    kubernetes.io/ingress.class: kong
username: jane
credentials:
  - jane-jwt
EOF
$ kubectl apply -f ./jane-consumer-jwt.yaml
```

Task: Build our JWT

```
$ export JANE_HEADER=`echo -n '{"alg":"RS256","typ":"JWT"}' | openssl base64 | tr
-d '=' | tr '/+' '_-' | tr -d '\n'`
$ export JANE_PAYLOAD=`echo -n '{"iss":"jane-issuer"}' | openssl base64 | tr -d '='
| tr '/+' '_-' | tr -d '\n'`
$ export JANE_HEADER_PAYLOAD=$JANE_HEADER.$JANE_PAYLOAD
$ export JANE_PEM=`cat ./jane.pem`
$ export JANE_SIG=`openssl dgst -sha256 -sign <(echo -n "${JANE_PEM}") <(echo -n
"${JANE_HEADER_PAYLOAD}") | openssl base64 | tr -d '=' | tr '/+' '_-' | tr -d '\n'`
$ export JANE_TOKEN=$JANE_HEADER.$JANE_PAYLOAD.$JANE_SIG
$ echo $JANE_TOKEN
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJqYW5lLWlzc3VlciJ9.u1olhZMrSiIdR64B6
JePrrLmq2nV1rSuYfPjhIw0xLPJD0C33Z3znffb4vrwoyQunlwLTcdC0WTGQhEcELGRqCFZxOiITMmSfWvv
O9Z5WYcyjmi9t6zAntqOuB4fJvDxLXSC9xKgFhYXh81zXyKmtdZwdJDuOrY8Lxkmdcgm-5yDUGd5dZPwgDk
0iIyNo61m6Wuk5j15EPmTR9XXWonqnshviiqW_p-Q9j6FXwTDrjLRS6cLKGaudAvrBB3vkPGPG21rKbAlXf
Xgi6MgBk7fXsAkEIqeX77P-H6pZFP9TkcyhCy1dSD-uvesW0lvMPvWr9111CItygk8Gskvpen_4g
$
```

Task: Inspect Token

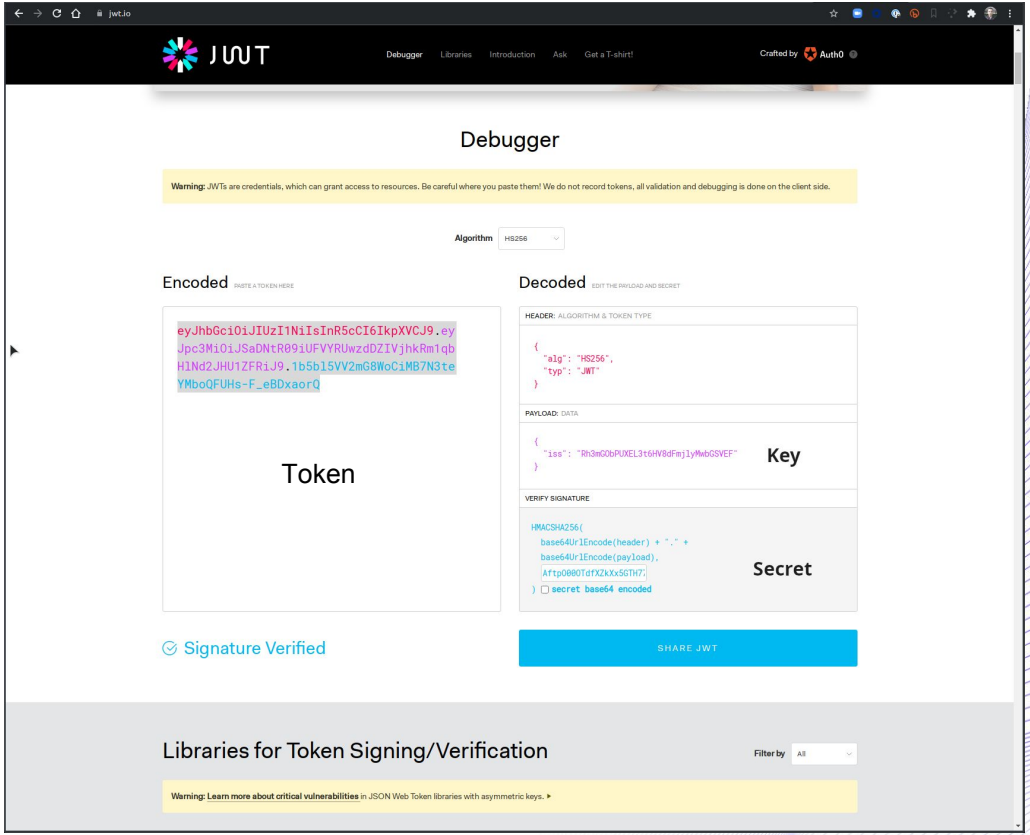
Navigate to http://jwt.io and use values generated for $TOKEN to see $KEY as payload, and $SECRET to verify the signature.

```
{"iss": "jane-issuer"}
```

```
WD6ynQrV8v6G12CI4tBv5p2Z8S8ZxDjV
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJBOUp4dzZzVGlSVTFROVVUU3pxcWtiQ1NwVjY2UkpuNiJ9.aSj7tiPXq6QJSzEswZwg3ByfJAO__I4xP4dhN7ULd4M
```

You can also use the values of $KEY and $SECRET to generate and sign a token.



Task: Consume the service with JWT credentials

Try consuming the service with no credentials:

```
$ http -h GET kongcluster:30000/httpbin
```

```
HTTP/1.1 401 Unauthorized
```

Consuming the service with JWT credentials:

```
$ http -h GET kongcluster:30000/httpbin Authorization:"Bearer $JANE_TOKEN"
```

```
HTTP/1.1 200 OK
```

Troubleshooting Tips/Tricks

- You can use JWT Debugger at https://jwt.io to generate a token or inspect its payload.
- You can also use jsonwebtokencli to encode and decode tokens.
- As noted, payload can be easily examined, so be careful about the information you include in it.
- Always use values that would mean nothing to another user, and only mean something to the consuming application.
- If authentication is failing, use `/consumers/<username>/jwt` endpoint to confirm that credentials/claims used in generating JWT are correct.

mTLS Plugin

What is Transport Layer Security (TLS)?

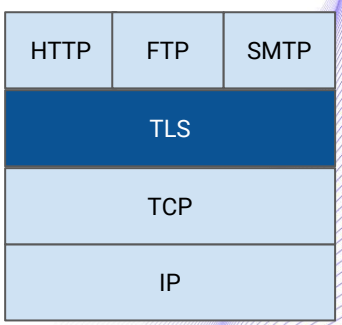
Before we talk about mTLS (Mutual TLS), let's explain what TLS (Transport Layer Security) is.

TLS is a cryptographic protocol used to provide communication security, privacy and data integrity between applications exchanging information over a computer network. It is widely used in applications such as email, IM and VOIP, however it is mostly known for use as security layer in HTTPS.

It is successor of the now-deprecated Secure Sockets Layer (SSL).

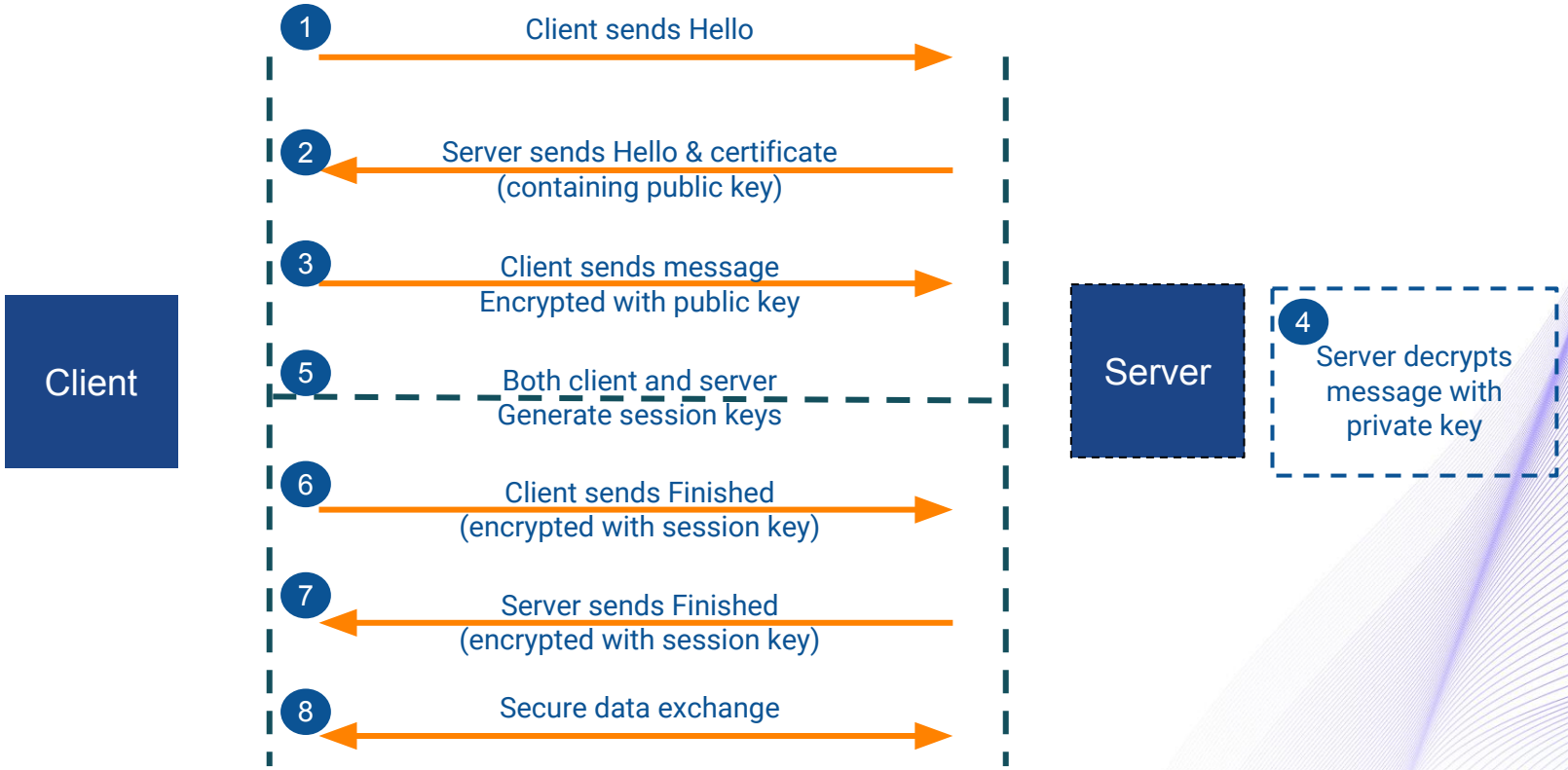
The typical TLS process works like this:

1. Client connects to server
2. Server presents its TLS certificate
3. Client verifies the server's certificate
4. Client and server exchange information over encrypted TLS connection



TLS lies between the TCP and presentation layers of the TCP/IP stack. By default the TLS protocol only proves the identity of the server to the client and the authentication of the client to the server is left to the application layer.

How does TLS work?



What is mTLS?

Mutual TLS (mTLS) is a process in which two parties establish an encrypted TLS connection using TLS protocol to authenticate each other.

With mTLS there are few more steps than with TLS:

1. Client connects to server
2. Server presents its TLS certificate
3. Client verifies the server's certificate
4. ***Client presents its TLS certificate***
5. ***Server verifies the client's certificate***
6. ***Server grants access***
7. Client and server exchange information over encrypted TLS connection

mTLS authentication is often used where a number of programmatic clients are connecting to specific web services and where security requirements are higher as compared to consumer environments. mTLS mitigates the risk of moving services to cloud by preventing malicious third parties from imitating genuine apps, using identities that are cryptographically verified.

Identity Validation & Certificate Authorities 101

- Kong needs the trust chain, and not just the CA, to validate certificates.
- The Certificate Authority (CA) Chain is a set of intermediate and root certificates used to establish the connection between a certificate and a Certificate Authority that issued the certificate.
- Intermediate certificates in the trust chain could be supplied by the client, or bundled with the CA certificate.
- The end-entity certificate is signed by the intermediate certificate (subordinate CA), which in its turn is signed by a root certificate, or another intermediate certificate signed by root.

What is the mTLS Plugin?

- Kong's Mutual TLS Authentication plugin implements mTLS authentication based on a client-supplied certificate and trusted CA list configured on Kong as follows
  - Kong will validate the client certificate passed in to determine if it is issued by the CA certificate that is associated with mTLS plugin
  - If the trust chain can be established, Kong will read the subject name on the client certificate and search consumer objects for a match
  - Once the consumer is found, Kong will allow the request and apply consumer-specific policies
- The plugin can be configured such that any client with a valid certificate can access the Service/API
- However, you can restrict usage to only some authenticated users by adding the ACL plugin and creating ‘allowed’ or ‘denied’ groups of users.
- The plugin can be enabled on a service, a route, or globally.

mTLS Plugin Usage

For a Consumer to authenticate it must provide a valid certificate and complete a mutual TLS handshake with Kong

The plugin validates certificate against the configured CA list

If the certificate is not trusted or has expired, the response will be

```
HTTP 401 TLS certificate failed verification.
```

If Consumer did not present a valid certificate then the response will be

```
HTTP 401 No required TLS certificate was sent
```

We will explore the mTLS plugin by way of a lab exercise

Lab: Install and Configure mTLS Plugin

In this lab you will:

1. **Create a self-signed CA Certificate, and a client certificate signed by this CA**
2. Add the self-signed CA Certificate to Kong as a trusted CA
3. Create two services (confidential/public) with corresponding routes and a consumer
4. Add the mTLS plugin to the confidential service
5. Apply consumer-level rate limiting
6. Verify Kong can proxy requests when consumer sends certificate
7. Verify that requests sent by different consumer types have different limits

Task: Delete the httpbin-demo Namespace

```
$ cd ~/edu-kgac-202/exercises/mtls
$ kubectl delete ns httpbin-demo
namespace "httpbin-demo" deleted
$
```

We will be deploying a couple custom ingresses to demonstrate the mtls on one path vs not having it on another.

Task: Create a self-signed certificate

The following script creates a self-signed CA Certificate, and a client certificate signed by this CA

```
$ cd ~/edu-kgac-202/exercises/mtls
$ ./create-certificate.sh
```

```
Generating RSA private key, 4096 bit long modulus (2 primes)
.....................................................................................................
TLS Web Client Authentication, E-mail Protection
            X509v3 Subject Alternative Name:
                email:demo@example.com
Certificate is to be certified until Sep 17 22:01:57 2022 GMT (375 days)

Write out database with 1 new entries
Data Base Updated
```

Here we are using openssl to create a client certificate - the actual details on the creation of this certificate is out of scope for this course.

We will associate this certificate with the confidential route in a future step

Kong Validating Client/Server Certificates

We have successfully created a self-signed CA certificate, however Kong is not aware of it, as it is locally stored in ~/.certificates/ folder.

This CA certificate needs to be uploaded to Kong and stored as a trusted CA, as its is used by Kong to verify the validity of a client certificate when handling encrypted requests with mTLS.

In this case the certificate associated with consumer is directly signed by the self-signed root certificate, so no intermediate certificates are present in the chain.

```
$ openssl crl2pkcs7 -nocrl -certfile ~/.certificates/client.crt \
| openssl pkcs7 -print_certs -noout
```

```
subject=C = WD, ST = Earth, O = Kong Inc., OU = Kong Academy, CN = example.com,
emailAddress = demo@example.com

issuer=C = WD, ST = Earth, L = Global, O = Kong Inc., CN = Kong CA
```

Task: Set up public and a private services & routes

We'll set up two new services and routes for the purposes of this lab - one public and one private:

```
$ kubectl apply -f ./httpbin-ingress.yaml
namespace/httpbin-demo created
service/confidential-service created
service/public-service created
ingress.networking.k8s.io/public-route created
ingress.networking.k8s.io/confidential-route created
$
```

Task: Verify traffic is being proxied

Check the services are configured correctly

```
$ http --verify=no GET https://kongcluster:30443/public
$ http --verify=no GET https://kongcluster:30443/confidential
```

Each should give you a `200 OK` response for each albeit with different output

```
HTTP/1.1 200 OK
...
```

Task: Implement the mTLS plugin to Kong

Now we'll enable the mTLS plugin for our confidential service

```
$ kubectl create secret generic httpbin-mtls -n httpbin-demo \
  --from-literal=id=cce8c384-721f-4f58-85dd-50834e3e733a \
  --from-file=cert=/home/labuser/.certificates/ca.cert.pem \
  -o yaml --dry-run=client > ./httpbin-mtls-secret.yaml
$ kubectl apply -f ./httpbin-mtls-secret.yaml
$ kubectl label secret httpbin-mtls -n httpbin-demo konghq.com/ca-cert='true'
$ kubectl annotate secret httpbin-mtls -n httpbin-demo \
  kubernetes.io/ingress.class=kong
$ kubectl apply -f ./httpbin-ingress-mtls.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-mtls unchanged
service/confidential-service configured
service/public-service unchanged
ingress.networking.k8s.io/public-route unchanged
ingress.networking.k8s.io/confidential-route configured
$
```

Task: Verify access for private service without a certificate

Let's consume the API on the private route without a certificate

```
$ http --verify=no https://kongcluster:30443/confidential
```

As expected access it not granted as no certificate was passed

```
{
    "message": "No required TLS certificate was sent"
}
```

Now let's issue a request while passing the certificate

Task: Create a consumer

As the certificate that we'll be using in the next steps will be associated with a consumer, let's create a consumer now

```
$ kubectl create secret generic mtls-consumer -n httpbin-demo \
  --from-literal=kongCredType=key-auth \
  --from-file=key=/home/labuser/.certificates/client.key \
  -o yaml --dry-run=client > ./mtls-consumer-secret.yaml
$ kubectl apply -f ./mtls-consumer-secret.yaml
secret/mtls-consumer created
$ kubectl apply -f ./mtls-consumer.yaml
kongconsumer.configuration.konghq.com/mtls-demo created
$
```

Task: Verify access for private service with a certificate

Now we'll consume our confidential service using the certificate associated with consumer

```
$ http --verify=no \
--cert=/home/labuser/.certificates/client.crt \
--cert-key=/home/labuser/.certificates/client.key \
https://kongcluster:30443/confidential
```

```
HTTP/1.1 200 OK
...

"HTTPie/0.9.8"
```

Here the `client.crt` is used to verify identify and `client.key` is used to encrypt the data.

Task: Verify public route is unaffected

Let's check the public route still works without certificates

```
$ http --verify=no GET https://kongcluster:30443/public
```

```
HTTP/1.1 200 OK
```

Task: Configure and Test Rate Limiting

If the client certificate is issued by the CA associated with the mTLS plugin, Kong will establish the trust chain, and attempt to match the subject alternative name on the client certificate with a consumer object. Once the consumer is found, Kong will allow the request and apply consumer-specific policies. To validate this, we will add the Rate Limiting plugin in this section & check that rate limiting plugin works with consumer identification using the client certificate.

```
$ kubectl apply -f ./mtls-consumer-rate-limiting.yaml
namespace/httpbin-demo unchanged
kongplugin.configuration.konghq.com/httpbin-rate-limiting created
kongconsumer.configuration.konghq.com/mtls-demo configured
$ (for ((i=1;i<=10;i++))
   do
     http -h --verify=no --cert=/home/labuser/.certificates/client.crt \
          --cert-key=/home/labuser/.certificates/client.key \
          https://kongcluster:30443/confidential \
          | head -1
    done)
```

Hardening the Kong Data Plane

Hardening the data plane

There are a few other specific security measures you should consider on your Kong Deployment, which are listed here for reference, but we will not go through each one individually.

- ✓ Health check for the Kong data plane
- ✓ Data plane mTLS certificate is managed by a certificate manager
- ✓ Save the encrypted keys for Key-authentication Encrypted Plugin in a Vault
- ✓ Disable Kong debug header
- ✓ Global observability plugins are enabled
- ✓ Global rate limit plugins are enabled

Summary

In this lesson we learnt how to

- Secure the data plane on Kong
- Deploy and configure the Rate limiting plugin
- Configure Authentication with the JWT and Mutual TLS plugins
- Identify some general Kong data plane hardening measures

Questions?

What's next?

In the next section we will look at the OIDC Plugin

Thank You