Search Guard

SSL

Official documentation for Search Guard 2 SSL

Version: 1.0

Overview

Search Guard® SSL is a plugin for Elasticsearch which provides SSL/TLS encryption and authentication. It offers:

- Node-to-node encryption on the transport layer (SSL/TLS)
- Secure REST layer through HTTPS (SSL/TLS)
- JDK SSL and Open SSL support
- Support for Kibana 4, logstash and beats

The only external dependency is Netty 4 (and Tomcat Native if Open SSL is used).

Search Guard SSL is the foundation layer of Search Guard 2, which offers authentication and authorization on top of Search Guard SSL. Search Guard is also available as an Elasticsearch plugin from floragunn. You can find the github repository for Search Guard 2 here.

If you just need to encrypt your traffic, and make sure only trusted nodes and clients can join and access a cluster, Search Guard SSL is all you require. If you need authentication and authorization, you need both Search Guard SSL + Search Guard 2.

In this document the following abbreviations are being used:

- ES: Elasticsearch
- SG SSL: Search Guard SSL
- SG: Search Guard 2

This documentation refers only to Search Guard SSL 2.0 and above. Search Guard for ES 1.x is not actively maintained anymore.

For the impatient

If you just want to get up and running quickly, please refer to the installation guide and the quickstart tutorial

Motivation

While Elasticsearch is often used for storing and searching sensitive data, it does not offer encryption or authentication/authorization out of the box.

In order to secure your sensitive data, the first step is to encrypt the entire traffic from and to your ES cluster via TLS. While it is possible to implement an authentication/authorization plugin without TLS encryption, we strongly believe that this will only get you “fake security”: An attacker can easily spy on your traffic, modify data and even join the cluster without permission.

By using TLS the traffic between ES nodes and ES clients will be encrypted. Which means that

- you can be sure that nobody is spying on the traffic

- you can be sure that nobody tampered with the traffic

However by using SSL certificates you can also make sure that **only identified and trusted nodes or clients** can interact with the cluster.

To make this efficient SG SSL can use Open SSL as the SSL implementation.

TLS in a nutshell

Note: We will explain how SG SSL uses TLS on the transport layer between nodes to encrypt the traffic, but it can also be used to encrypt the traffic on the REST layer. The principles are the same for both layers. While it is possible to use TLS on only one layer, you need to enable it for both transport and REST layer to get proper security.

TLS (often referred to as “SSL” for historical reasons) is a cryptographic protocol to secure the communication over a computer network. It uses **symmetric encryption** to encrypt the transmitted data, by negotiating a **unique secret** at the start of any **TLS session** between two participating parties. It uses **public key cryptography** to authenticate the identity of the participating parties.

TLS supports many different ways for exchanging keys and encrypting data, but that is beyond the scope of this document. We’ll focus only on the main concepts necessary to set up SG SSL.

Certificates & Certificate authorities

TLS uses certificates (“digital certificates”) for encrypting traffic and for identifying and authenticating the two participating parties in an TLS session. In a typical client/server scenario, only the identity of the server is verified. However, TLS is not limited to that, and we’ll use client authentication later on in order to verify that only trusted clients can access ES.

To be more precise,

> a digital certificate certifies the ownership of a public key by the named subject of the certificate.
>
> (Source: Wikipedia).

The public key is then used for the actual encryption.

Trust Hierarchies

A certificate can be used to issue and sign other certificates, building a trust hierarchy amongst these certificates. The certificates can be **intermediate root certificates** or **end entity certificates**.

A **root certificate**, issued by a **root certificate authority (root CA)** is at root of this hierarchy. A certificate authority represents a **trusted third party** authority, trusted both by the owner of the certificate and communication partner. All major browser come with a pre-installed list of trusted certificate authorities, like Thawte, GlobalSign, DigiCert or Comodo.

A root CA uses its own certificate to issue other certificates. An intermediate root certificate can be used to issue other certifictes, while an end entity certificate is intended to be installed directly on a server directly. With an end entity certificate do you not sign other certificates.

If a TLS session is to be established, all certificates in the hierarchy are verified, up to the root certificate. The communication partner is only trusted if each certificate in this chain is trusted.

SSL handshake

Before any user data is sent between these parties, first a **TLS session** has to be established. This process is called **TLS handshake**.

The details of this handshake can vary slighty, depending on the selected cipher suite, but at a very high level the following steps are performed:

- The client sends a ClientHello message, containing the information which TLS version and cipher protocols it supports
- The server sends a ServerHello message, containng the choosen protocol version and cipher suite
- The server sends its certificate, containing the public key
- The client verifies the certificate and all intermediate certificates against its list of installed root CAs
- If the client cannot verify the certificate, it aborts the connection or displays a warning, depending on the type and the settings of the client
- If the certificate could be verified, the client sends a “PreMasterSecret” to compute a common secret, called the “master secret”. This master secret is used for encrypting the traffic. The PreMasterSecret is encrypted using the public key of the server certificate.
- The server tries to decrypt the clients message using its private key. If this fails, the handshake is considered as failed
- If decryption succeeds, the handshake is complete, and all traffic is encrypted using the master secret.

While at this point the servers identity is verified, the clients identity is not, since it never sent any information regarding its own identity. But this is possible too and it is called **Client Authentication** (aka **Mutual authentication** or **two-way authentication**) and is an optional part of the handshake protocol. If the server is configured to use client authentication, the following steps are added in the handshake:

- The server sends a CertificateRequest message to the client
- The client sends its own certificate, containing the clients public key, to the server
- The server verifies the certificate using its own installed list of root CAs and intermediate CA
- If thus succeeds, the connection is mutually authenticated

Note that this is a very simplified version of the handshake protocol, only to demonstrate the basic principles behind it. In short: The handshake protocol uses a combination of certificates and public/private keys to verify the identities of the communication partners and to negotiate a master secret for encrypting the traffic.

Keystore and Truststore

Java uses **keystores** and a **truststores** to store certificates and private keys. The truststore contains all trusted certificates, typically root CAs and intermediate certificates. It is used to **verify credentials** from a communication partner.

The **keystore** holds the private keys and the associated certificates. It is used to **provide credentials** to the communication partner.

SG SSL supports two key-/truststore formats: JKS and PKCS12.

In a typical SG SSL setup, each node in the cluster has both a keystore and a truststore. If a node wants to communicate with another node, it uses its own certificate stored in the keystore to identify itself.

The other node uses the root CAs stored in the truststore to verify the other nodes certificate.

This of course is performed in both ways, because a node acts as a "client" and "server" at the same time: It sends requests to other nodes, acting as a client, and receives request, acting as a server.

Minimal setup

A minimal SG SSL setup might look like this:

Root CA

You need a root CA in order to create the required certificates for the nodes. If you already have a PKI infrastructure in place, you're good to go. Otherwise, you need to create a root CA first. SG SSL comes with scripts to help you set up your root CA.

Generate certificates

Using the root CA, you then generate certificates for all participating nodes (and optionally clients, if you use client authentication) in your setup. These certificates are signed with your root certificate

Generate keystore and truststore files

For each node, you create a keystore and a truststore file. The keystore contains the nodes own certificate, while the truststore contains the root CA. The truststore can be the same on all nodes, while the keystore should be different, since each node should have its own, exclusive certificate

Configuring SG SSL

Add the configuration settings of SG SSL to the elasticsearch configuration, adjust it to your needs, and start the node.

Note: Technically speaking, a Root CA is not really mandatory. You could as well just distribute the certificates of all nodes to the truststore of all nodes. While this would work, it's highly impractical. That is why we are not dig into this scenario any deeper.

Installing SG SSL

Prerequisites

- Java 7 or 8 (Oracle Java 8 recommended)
- Elasticsearch 2.0 or above
- Optional: Tomcat Native and OpenSSL

Disable the security manager

If you are running Elasticsearch 2.0.x or 2.1.x, then you have to disable the security manager in elastic-search.yml. For Elasticsearch >= 2.2.x this is no longer necessary.

In order to disable the security manager, add the following line to your elastcsearch.yml configuration file:

```
security.manager.enabled=false
```

Install SG SSL

Choosing the right version

The versioning schema of SG SSL is: e1.e2.e3.sgv

where:

- e1: Elasticsearch Major Version
- e2: Elasticsearch Minor Version
- e2: Elasticsearch Fix Version
- sgv: Search Guard SSL Version

The following table illustrated which SG SSL version you need for a particular ES version:

ES version	SG SSL version
1.x.y	not supported
2.0	not supported
2.0.1	not supported
2.0.2	2.0.2.5
2.1.0	2.1.0.5
2.1.1	2.1.1.5
2.1.2	2.1.2.5
2.2.0	2.2.0.6
2.2.1	2.2.1.7
2.3.1	2.3.1.8
2.3.2	2.3.2.9

For an up-to-date version matrix, you can always refer to the Wiki on github

Install the plugin

Before installing the plugin, stop your ES node(s) if necessary.

From Maven

SG SSL can be installed it like any other Elasticsearch plugin. Change to the installation directory of ES, and execute:

```
bin/plugin install com.floragunn/search-guard-ssl/<version>
```

"<version>" is for example 2.3.2.9 (NOT v2.3.2.9)

Example:

```
bin/plugin install com.floragunn/search-guard-ssl/2.3.2.9
```

the following warning message by typing "y" (since ES >= 2.2)

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@     WARNING: plugin requires additional permissions     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
* java.lang.RuntimePermission accessClassInPackage.sun.misc
* java.lang.RuntimePermission getClassLoader
* java.lang.RuntimePermission loadLibrary.*
* java.lang.reflect.ReflectPermission suppressAccessChecks
* java.security.SecurityPermission getProperty.ssl.KeyManagerFactory.algorithm
See http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html
for descriptions of what these permissions allow and the associated risks.
```

Alternatively, you can download the **zip** from Maven Central and install it offline by typing:

```
bin/plugin install file:///path/to/search-guard-ssl-<version>.zip
```

Next steps

After performing these steps, SG SSL is installed. Next:

- Create and install the SSL certificates on each node
- see chapters “Quickstart” and “Certificates”
- Configure SG SSL
- see chapter “Configuration”
- Start your nodes and enjoy the added security

Quick Start

A note on security

This tutorial shows how to quickly set up SG SSL. Note that the settings and passwords we use here are **not safe for production**, and only meant to get SG SSL working as quickly as possible!

Install the plugin

First, install SG SSL for your particular ES version on each node. For installation instructions and to figure out which SG version you need for your Elasticsearch installation, please refer to the chapter Installation.

Generating the keystore and truststore

For SSL to work, you have to have a **keystore** and a **truststore** containing all required certificates and keys on each node. SG comes with scripts that will generate all these required files for you. The scripts have been tested on Linux and OSX.

In particular, the scripts will generate a truststore file containing a generated root certificate. This truststore file can be used on all nodes equally and has to be copied to the `config` directory on all nodes manually.

The script will also create three keystore files. Each node has to have its own, dedicated keystore file, since it contains the nodes own certificate plus its private key. For each node, copy one of the generated keystore files to the `config` directory manually.

In addition, certificates for the REST layer to be installed on your browser are also generated. These cetificates are only needed if you enable client authentication in the SG SSL configuration.

The script uses OpenSSL for generating all required artifacts. If you do not have OpenSSL already installed on your machine, please do so. If you cannot use OpenSSL on your machine, you'll need to find some other ways to obtain the required files. In this case, lease refer to the chapter Certificates.

In order to find out if you have OpenSSL installed, open a terminal and type

```
openssl version
```

If installed, this should print out the version number of your OpenSSL installation.
Make sure its at least version 1.0.1k.

In order to generate the required artifacts, please execute the following steps:

Download SG SSL or clone the repository

In order to obtain and run the scripts, you need to download the SG SSL source code onto your machine.

If you have git installed on your machine, open a terminal, and change to the directory where you want to download SG SSL. Type:

```
git clone https://github.com/floragunncom/search-guard-ssl.git
```

You should see something like:

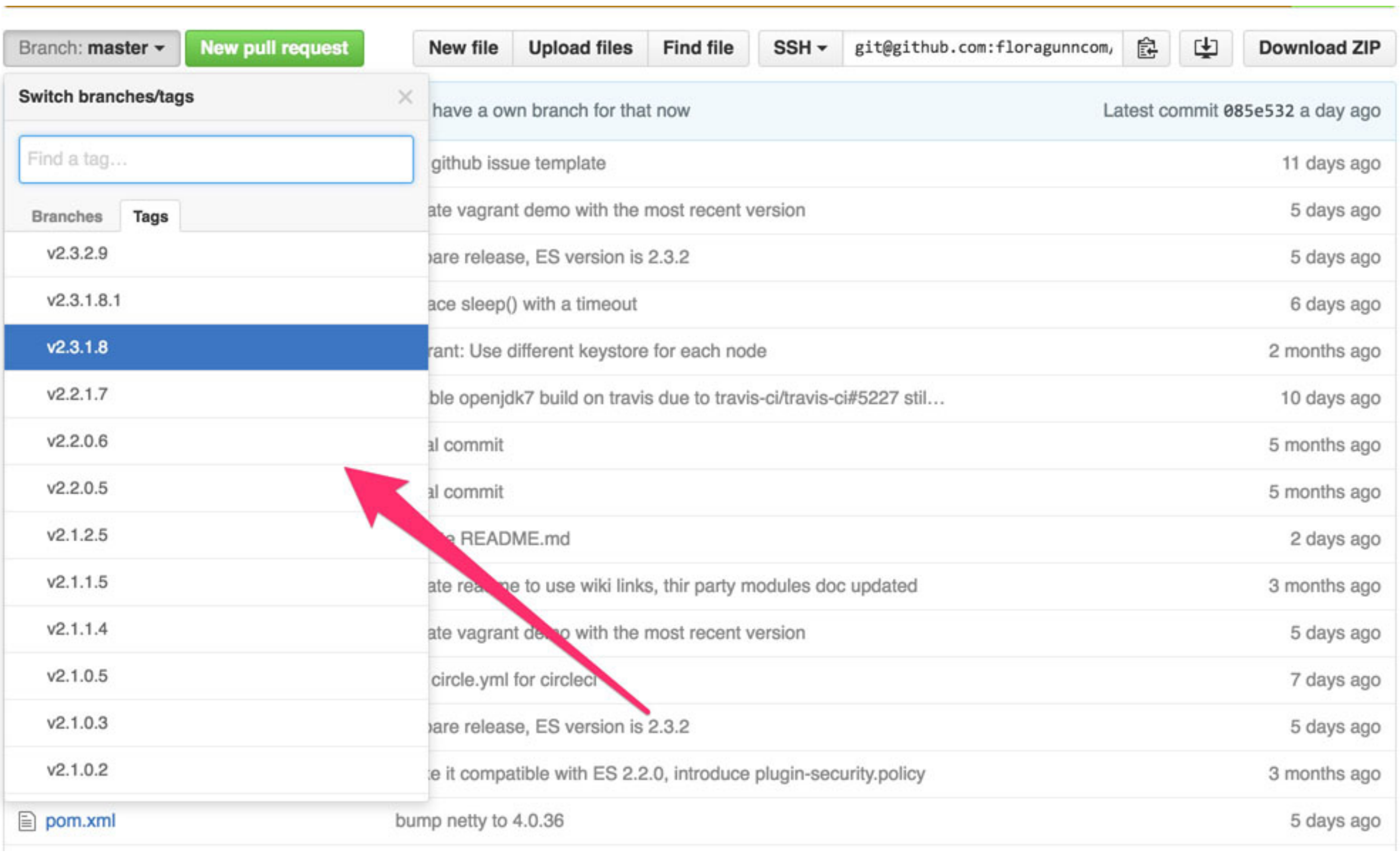
```
Cloning into 'search-guard-ssl'...
```

on the command line. The repository is now being downloaded on your computer.

If you do not have git installed, and do not want to install it, you can also download the source files as a zip archive directly from github. First, visit the repository at this URL:

```
https://github.com/floragunncom/search-guard-ssl
```

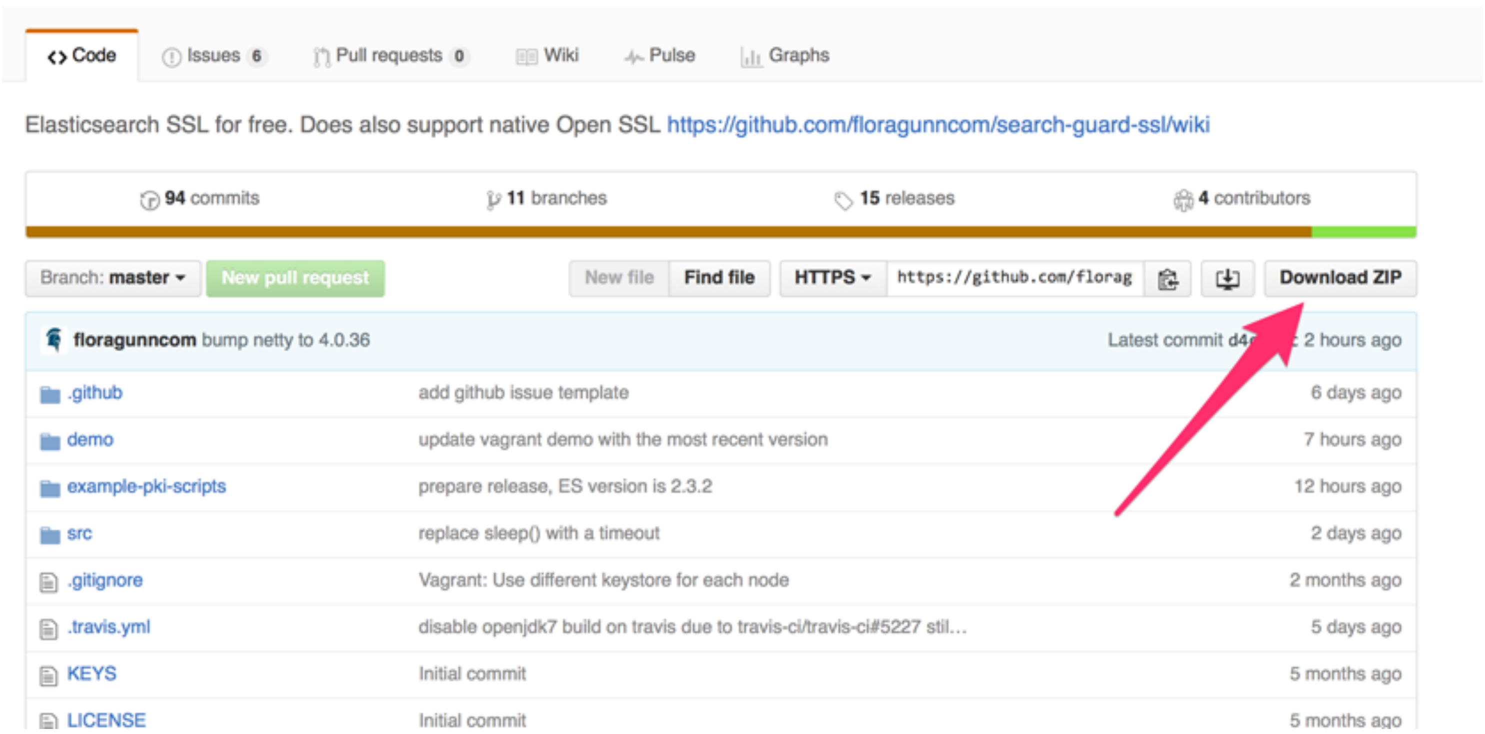
Choose the version to download:



have a own branch for that now — Latest commit 085e532 a day ago
github issue template — 11 days ago
ate vagrant demo with the most recent version — 5 days ago
are release, ES version is 2.3.2 — 5 days ago
ace sleep() with a timeout — 6 days ago
rant: Use different keystore for each node — 2 months ago
ble openjdk7 build on travis due to travis-ci/travis-ci#5227 stil… — 10 days ago
al commit — 5 months ago
al commit — 5 months ago
README.md — 2 days ago
ate rea me to use wiki links, thir party modules doc updated — 3 months ago
ate vagrant de o with the most recent version — 5 days ago
circle.yml for circleci — 7 days ago
are release, ES version is 2.3.2 — 5 days ago
e it compatible with ES 2.2.0, introduce plugin-security.policy — 3 months ago
pom.xml — bump netty to 4.0.36 — 5 days ago

In order to figure out which SG SSL version you need for your particular ES version, refer to the chapter “Installation”, or visit the version matrix at our github Wiki

Use the “Download as zip” button to download the archive:



Elasticsearch SSL for free. Does also support native Open SSL https://github.com/floragunncom/search-guard-ssl/wiki

floragunncom bump netty to 4.0.36 Latest commit d4… 2 hours ago

.github	add github issue template	6 days ago
demo	update vagrant demo with the most recent version	7 hours ago
example-pki-scripts	prepare release, ES version is 2.3.2	12 hours ago
src	replace sleep() with a timeout	2 days ago
.gitignore	Vagrant: Use different keystore for each node	2 months ago
.travis.yml	disable openjdk7 build on travis due to travis-ci/travis-ci#5227 stil…	5 days ago
KEYS	Initial commit	5 months ago
LICENSE	Initial commit	5 months ago

Save and unzip the archive in a directory of your choice.

Execute the example script

Open a terminal and cd into the directory where you downloaded or extracted the SG SSL source code to. You'll find a folder called `example-pki-scripts`. Change to this folder.

The script we need to execute is called `./example.sh`. Make sure you have execute permissions on this file (chmod the permissions if needed) and execute it. All required artifacts are now generated. If execution was successful, you'll find a couple of generated files and folders inside the `example-pki-scripts` folder.

If for any reason you need to re-execute the script, execute `./clean.sh` in the same directory first. This will remove all generated files automatically.

Copying the keystore and truststore files

Now that all required artifacts are in place, we need to copy them to Elasticsearch.

Amongst others, the script generated a file called `truststore.jks`. Copy this file to the `config` directory of all ES nodes.

The script also generated three keystore files:

- `node-0-keystore.jks`
- `node-1-keystore.jks`
- `node-2-keystore.jks`

The keystore files are specific per node. Copy `node-0-keystore.jks` to the `config` directory of your first ES node, `node-1-keystore.jks` to the second and so forth.

The config directory of your first ES node should now look like:

```
elasticsearch-2.2.0
|
└─── config
    |   elasticsearch.yml
    |   logging.yml
    |   node-0-keystore.jks
    |   truststore.jks
    ├─── scripts
    |    |   ...
    | ...
```

Configuring the plugin

SG SSL is configured in the `config/elasticsearch.yml` file of your ES installation. Stop any running ES node and add the following lines to this file. It does not matter where in the config file you add them, so we'll just append them to the end.

```
searchguard.ssl.transport.keystore_filepath: node-0-keystore.jks
searchguard.ssl.transport.keystore_password: changeit
searchguard.ssl.transport.truststore_filepath: truststore.jks
searchguard.ssl.transport.truststore_password: changeit
searchguard.ssl.transport.enforce_hostname_verification: false
```

This has to be done for all nodes of your cluster. Note that you have to adjust the name of the keystore file (`node-0-keystore.jks` in this example) for each node separately. So on your first ES node you'd use

```
searchguard.ssl.transport.keystore_filepath: node-0-keystore.jks
```

And on the second node:

```
searchguard.ssl.transport.keystore_filepath: node-1-keystore.jks
```

And so on. While it **is** possible to use the same keystore file on each node, we recommend installing a seperate file on each node, because this is closer to a production setup.

Testing the installation

Your nodes are now ready to talk SSL to each other! Just start ES as normal, and watch the logfile. The nodes should start up without error. You can safely ignore the following infos and warnings in the logfile:

```
INFO: Open SSL not available because of java.lang.ClassNotFoundException:
 org.apache.tomcat.jni.SSL
```

This simply means that you use JCE (Java Cryptography extensions) as your SSL implementation. On startup, SG looks for OpenSSL support on your system. Since we have not installed it yet, SG falls back to the built-in Java SSL implementation.

```
WARN: AES 256 not supported, max key length for AES is 128.
To enable AES 256 install 'Java Cryptography Extension (JCE)
Unlimited Strength Jurisdiction Policy Files'
```

If you use Oracle JDK, the length of the cryptographic keys is is limited for judical reasons. You'll have to install the Java Cryptography Extension (JCE)
Unlimited Strength Jurisdiction Policy Files to use longer keys. For our quickstart tutorial, you can ignore this warning for the moment.

Congrats. Your ES nodes now talk TLS to each other on the transport layer!

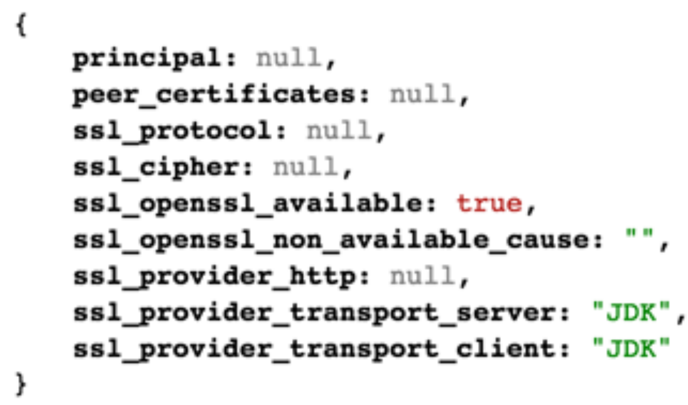
We have not configured HTTPS for the REST-API yet, so you can still access ES via a browser (or curl for that matter) as usual by typing

```
http://127.0.0.1:9200/
```

This should give you some information about ES in JSON format. You can also display some configuration information from SG SSL directly by visiting:

```
http://127.0.0.1:9200/_searchguard/sslinfo?pretty
```

Which should display something like this:



```
{
    principal: null,
    peer_certificates: null,
    ssl_protocol: null,
    ssl_cipher: null,
    ssl_openssl_available: true,
    ssl_openssl_non_available_cause: "",
    ssl_provider_http: null,
    ssl_provider_transport_server: "JDK",
    ssl_provider_transport_client: "JDK"
}
```

Configuring HTTPS

Now that the traffic between the nodes is TLS-encrypted, we want to make sure that also the traffic via the REST-API is secure. Simply speaking, a client using the REST-API must be configured in a similar way as the participating nodes. In our case, the client will be a browser, but you can also use curl.

A note on using curl: Since the generated root certificate is self-signed, it is not trusted by default. When accessing ES via a browser, this is not a problem: The browser will give you a warning and the possibility to import the Root CA. When using curl, you need to add the `--insecure` flag in order to skip the certificate validation. Otherwise, curl will fail with a certificate validation error.

There is one difference though: We said earlier that each node has to authenticate itself in order to being able to join the cluster. For a browser talking HTTPS to ES this so called “client authentication” is optional. We will set it up later.

In order to activate and configure HTTPS, stop any running nodes and add the following lines to the end of your `config/elasticsearch.yml` file of your ES installation on each node:

```
searchguard.ssl.http.enabled: true
searchguard.ssl.http.keystore_filepath: node-0-keystore.jks
searchguard.ssl.http.keystore_password: changeit
searchguard.ssl.http.truststore_filepath: truststore.jks
searchguard.ssl.http.truststore_password: changeit
```

You’ll notice that this configuration is nearly identical with the transport layer configuration we did before. While you can use different certificates for the transport layer and the HTTPS layer, we’ll just use the same certificates for our tutorial.

Now start your node(s) and try to connect with HTTP first:

Note: The generated certificates are valid for the IP `127.0.0.1` only. So, in the following examples, do not use `localhost`, but `127.0.0.1` instead. If you use localhost, you’ll see an error message in the browser like: “The certificate is only valid for the following names: node-0.example.com, 127.0.0.1 …”

```
http://127.0.0.1:9200/
```

Since we configured SG SSL to only accept SSL/TLS connections, you should see an error messages in the browser and also in the logfiles. This is expected and means that SG SSL rejects all non-SSL connections.

Now try with HTTPS:

```
https://127.0.0.1:9200/
```

This should give you the a warning from the browser regarding our self signed certificate. Since we generated all certificates ourself, the browser does not trust the root CA and informs you about that. You can either ignore the warning and accept the unknown certificate. Or, import the root CA the script generated in the browser. Both appoaches differ from browser to browser and OS to OS.

Importing the Root CA

In order to make your browser trust our generated certificates, you need to import the Root CA, and, for some browsers or OS, additionally trust this certificate.

How this is done varies. For example, Firefox has it’s own list of trusted CAs. You can find them under Settings -> Advanced -> Certificates -> Show Certificates. Chrome on OSX uses the operating systems keychain. Please refer to your browser and/or OS documentation to find out how to import Root CAs on your particular system.

You will find the certificate to import in the directory `example-pki-scripts/ca`. Import the certificate named `root-ca.crt`. If you now access

```
https://127.0.0.1:9200/
```

The warnings should be gone. Congratulations. Your complete ES communication is encrypted now!

Optional: Client authentication

While it is common for HTTPS that only the servers identity is verified, SSL is not limited to that. This means that you can configure SG SSL to only accept HTTPS connections from trusted clients. In our example, from trusted browsers. By doing so, you can set up an authentication schema solely based on certificates. While this is far from sophisticated authentication/authorization based on groups and access rights, it might be already sufficient for your use case.

First, let's enable client authentication. Add the following line to the `config/elasticsearch.yml` file of your ES installation on each node:

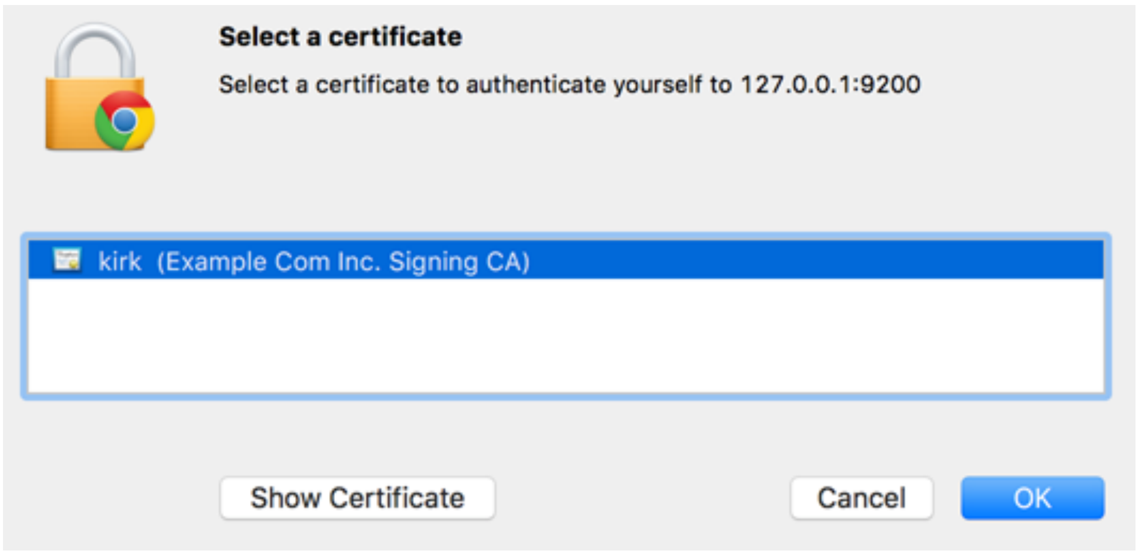
```
searchguard.ssl.http.clientauth_mode: REQUIRE
```

After restarting the node(s), try again to connect via your browser. You should see an error message like "Certificate-based authentication failed".

This means that SG SSL is asking your browser to identify itself. Since we have not installed any certificate for that purpose so far, the client cannot do so and SG SSL rejects the connection.

Similar to importing the Root CA, we now need to install a certificate that your ES nodes trust. The `example.sh` script also generated those for us. The certificates are called `kirk` and `spock`, and have been generated using the same root CA as for the node certificates. The client certificates have been generated in different formats. Which one you need to use again depends on your browser and OS.

After importing either one of the certificates, try to connect to ES again with your browser. This time, the browser asks you which certificate you want to us to identify yourself:



Select a certificate

Select a certificate to authenticate yourself to 127.0.0.1:9200

kirk (Example Com Inc. Signing CA)

Choose "kirk" or "spock", depending on which one you have installed, and click on ok. The connection should succeed, and you should again see the ES status information in JSON format.

If you visit the SG SSL info page again by entering:

```
https://127.0.0.1:9200/_searchguard/sslinfo?pretty
```

You should now see an output similar to this:

```
{
    principal: "CN=kirk,OU=client,O=client,L=Test,C=DE",
    peer_certificates: "3",
    ssl_protocol: "TLSv1.2",
    ssl_cipher: "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
    ssl_openssl_available: false,
    ssl_openssl_version: -1,
    ssl_openssl_version_string: null,
    ssl_openssl_non_available_cause: "java.lang.ClassNotFoundException: org.apache.tomcat.jni.SSL",
    ssl_provider_http: "JDK",
    ssl_provider_transport_server: "JDK",
    ssl_provider_transport_client: "JDK"
}
```

Note the `principal` entry, which displays some information about the client certificate you used to identify yourself to ES.

At this point, no client without a valid certificate can connect to ES.

Congratulations, your ES setup is now secured via SG SSL. No unauthorized client can connect, and the traffic is secured against sniffing and tampering.

Certificates

As described in the quickstart tutorial, you can always use our `example.sh` script to generate a root CA and all required certificates. The `example.sh` basically just executes three other scripts:

- `gen_root_ca.sh`
- Generates the root CA
- `gen_node_cert.sh`
- Generates a certificate for one node
- `gen_client_node_cert.sh`
- Generates a client certificate

Depending on your needs, you can of course execute each script manually, or use OpenSSL commands directly. If you already have a PKI infrastructure in place, please ask the respective authority in your company to create the required assets.

In case there is no PKI/Root CA established in your company yet, don't worry. You can easily build your own as described below.

The following section asumes you have OpenSSL installed on your machine. Execute all commands from inside the `example-pki-scripts` folder.

Creating a Root CA

The configuration settings for the Root CA are located in the `example-pki-scripts/etc` folder. You need to adjust two files:

- `root-ca.conf`
- `signing-ca.conf`

You should revise especially this section:

```
[ ca_dn ]
0.domainComponent         = "com"
1.domainComponent         = "example"
organizationName          = "Example Com Inc."
organizationalUnitName    = "Example Com Inc. Root CA"
commonName                = "Example Com Inc. Root CA"
```

And change the example values according to your needs.

In order to generate the root CA, execute the script `example-pki-scripts/gen_root_ca.sh` like:

```
./gen_root_ca.sh capassword_use_a_strong_one truststorepassword
```

The script expects two paramaters, the CA password and the truststore password.

If you're planning to use the generated artifacts for production, make sure to use strong passwords!

Now you get a bunch of files and you're now able to sign certificate signing requests (CSR) with you newly generated root CA. In other words, you can now start generating certificates for each node, and sign them with your root CA.

Creating the truststore

If you used the `example-pki-scripts/gen_root_ca.sh`, it already created a ready-to-use truststore file for you.

If you created the root certificate (chain) some other way, or if your company already has a root certificate (chain) in place, you can create the truststore manually. Get your root certificate chain (this means the root certificate itself and intermediate signing certificates if they exist) in PEM format and execute the following command:

```
keytool  \
    -importcert \
    -file chain-ca.pem  \
    -keystore truststore.jks   \
    -storepass mysecret_ts_passwd  \
    -noprompt -alias root-ca
```

In both cases, distribute the generated `truststore.jks` file to the `config` directory of all participating nodes of your cluster.

Creating and signing CSRs

A certificate signing request is a block of encrypted text, usually, but not always, generated on the server where you plan to use the certificate. It contains information about your organization, and the public key to be used with the certificate.

If you want to obtain a certificate you can use on a server, or, in this case, on a ES node, you:

- create a CSR
- send the CSR to a root CA
- get back a signed certificate

This needs to be done for each node separately.

For a tutorial how to create CSRs, please see here: https://www.digicert.com/csr-creation.htm.

You need to create a CSR for each node in your cluser. In the following "NODE_NAME" stands for the name of your ES node. You can pick any name you like here, for example `mycompany-elk-node-1`.

If you are familiar with OpenSSL you can use the following command directly to generate a CSR and private key:

```
openssl req -new -keyout NODE_NAME.key -out NODE_NAME.csr
```

can also use Javas keytool to generate keys and CSRs:

Generate a new key:

```
keytool -genkey \
        -alias     NODE_NAME \
        -keystore  NODE_NAME-keystore.jks \
        -keyalg    RSA \
        -keysize   2048 \
        -sigalg SHA256withRSA \
        -validity  712 \
        -keypass mykspassword \
        -storepass mykspassword \
        -dname "CN=nodehostname.example.com, OU=department, O=company, L=localityName,
C=US" \
        -ext san=dns:nodehostname.example.com
```

Generate a CSR (Certificate signing request):

```
keytool -certreq \
        -alias      NODE_NAME \
        -keystore   NODE_NAME-keystore.jks \
        -file       NODE_NAME.csr \
        -keyalg     rsa \
        -keypass mykspassword \
        -storepass mykspassword \
        -dname "CN=nodehostname.example.com, OU=department, O=company, L=localityName,
C=US" \
        -ext san=dns:nodehostname.example.com
```

important part here is CN=nodehostname.example.com This has to be the full qualified hostname of your node, or the ip address if no DNS entry exists.

You can now sign it with your root CA by using the following command:

```
openssl ca \
    -in NODE_NAME.csr \
    -notext \
    -out signed-NODE_NAME.csr.pem \
    -config etc/signing-ca.conf \
    -extensions v3_req \
    -batch \
    -passin pass:capassword_use_a_strong_one \
    -extensions server_ext
```

Alternatively, send the CSR to someone in your organization who can sign it with the root CA.

Creating the keystore

You can now import the signed certificates along with the intermediate certificates (if any) to the keystore.

```
cat ca/chain-ca.pem NODE_NAME-signed.pem | keytool \
    -importcert \
    -keystore NODE_NAME-keystore.jks \
    -storepass mykspassword \
    -noprompt \
    -alias NODE_NAME
```

Distribute `NODE_NAME-keystore.jks` to the appropriate node and put it into the config/ folder

Further reading

- https://www.digitalocean.com/community/tutorials/java-keytool-essentials-working-with-java-keystores
- https://tomcat.apache.org/tomcat-8.0-doc/ssl-howto.html

Enabled SSL ciphers and protocols

By default the following SSL cipher suites and protocols are enabled:

```
//https://wiki.mozilla.org/Security/Server_Side_TLS
"TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
"TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",
"TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
"TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",
"TLS_DHE_RSA_WITH_AES_128_GCM_SHA256",
"TLS_DHE_DSS_WITH_AES_128_GCM_SHA256",
"TLS_DHE_DSS_WITH_AES_256_GCM_SHA384",
"TLS_DHE_RSA_WITH_AES_256_GCM_SHA384",
"TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256",
"TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256",
"TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
"TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA",
"TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384",
"TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384",
"TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
"TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA",
"TLS_DHE_RSA_WITH_AES_128_CBC_SHA256",
"TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
"TLS_DHE_DSS_WITH_AES_128_CBC_SHA256",
"TLS_DHE_RSA_WITH_AES_256_CBC_SHA256",
"TLS_DHE_DSS_WITH_AES_256_CBC_SHA",
"TLS_DHE_RSA_WITH_AES_256_CBC_SHA"
"TLSv1.1"
"TLSv1.2"
```

When OpenSSL is used then `SSLv2Hello` is also enabled

Configuration

All configuration settings for SG SSL need to be placed in `config/elasticsearch.yml`. The SG SSL repository contains a configuration template (`searchguard-ssl-config-template.yml`) with all configuration options available. It's located at the root of the repository.

Transport layer SSL

Enabling and disabling transport layer SSL

searchguard.ssl.transport.enabled: (true/false)

Enable or disable node-to-node ssl encryption. (Optional, default: true)

Keystore

searchguard.ssl.transport.keystore_type: (PKCS12/JKS)

The type of the keystore file, JKS or PKCS12 (Optional, default: JKS)

searchguard.ssl.transport.keystore_filepath: keystore_node1.jks

Path to the keystore file, relative to the `config/` directory (mandatory)

searchguard.ssl.transport.keystore_alias: my_alias

Alias name (Optional, default: first alias which could be found)

searchguard.ssl.transport.keystore_password: changeit

Keystore password (default: changeit)

Truststore

searchguard.ssl.transport.truststore_type: PKCS12

The type of the truststore file, JKS or PKCS12 (default: JKS)

searchguard.ssl.transport.truststore_filepath: truststore.jks

Path to the truststore file, relative to the `config/` directory (mandatory)

searchguard.ssl.transport.truststore_alias: my_alias

Alias name (default: first alias which could be found)

searchguard.ssl.transport.truststore_password: changeit

Truststore password (default: changeit)

Hostname verification

searchguard.ssl.transport.enforce_hostname_verification: true

Enforce hostname verification (default: true).

searchguard.ssl.transport.resolve_hostname: true

If hostname verification is enabled, specify whether the hostname should be resolved against DNS (default: true).

Open SSL

searchguard.ssl.transport.enable_openssl_if_available: false

Use native Open SSL instead of JDK SSL if available (default: true)

HTTP/REST layer SSL

Enabling and disabling HTTP/REST layer SSL

searchguard.ssl.http.enabled: true

Enable or disable REST layer security (https), (default: false)

Keystore

searchguard.ssl.http.keystore_type: PKCS12

The type of the keystore file JKS or PKCS12 (default: JKS)

searchguard.ssl.http.keystore_filepath: keystore_https_node1.jks

Path to the keystore file, relative to the `config/` directory (mandatory)

searchguard.ssl.http.keystore_alias: my_alias

Alias name (default: first alias which could be found)

searchguard.ssl.http.keystore_password: changeit

Keystore password (default: changeit)

Truststore

searchguard.ssl.http.truststore_type: PKCS12

The type of the truststore file JKS or PKCS12 (default: JKS)

searchguard.ssl.http.truststore_filepath: truststore_https.jks

Path to the truststore file, relative to the `config/` directory (mandatory)

searchguard.ssl.http.truststore_alias: my_alias

Alias name (default: first alias which could be found)

searchguard.ssl.http.truststore_password: changeit

Truststore password (default: changeit)

OpenSSL

searchguard.ssl.http.enable_openssl_if_available: false

Use native Open SSL instead of JDK SSL if available (default: true)

Client authentication

searchguard.ssl.http.clientauth_mode: REQUIRE

Do the clients (typically the browser or the proxy) have to authenticate themself to the http server? To enforce authentication use REQUIRE, to completely disable client certificates use NONE, to use authentication when a certificate is availabl on the browser, use OPTIONAL. Default is OPTIONAL.

OpenSSL setup

Search Guard SSL can use Open SSL as the SSL implementation. This will result in better performance and better support for strong and modern cipher suites. With Open SSL its also possible to use strong chipers without installing Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files.

To enable native support for Open SSL follow these steps:

- Install latest OpenSSL version on every node (make sure its at least version 1.0.1k.)
- https://www.openssl.org/community/binaries.html
- Install Apache Portable Runtime (libapr1) on every node
- https://apr.apache.org
- On Ubuntu, Apache Portable Runtime can be installed with `sudo apt-get install libapr1`
- Download netty-tcnative 1.1.33.Fork15 .jar for **your** platform
- http://repo1.maven.org/maven2/io/netty/netty-tcnative/1.1.33.Fork15/version, where version is one of `_linux-x86.jar_`, `_64-fedora.jar_`, `_osx-x86_64.jar_` or `_windows-x86_64.jar_`
- Put it into the elasticsearch `plugins/searchguard-ssl/` folder (on every node of course)
- If you update the plugin (or re-install it after removal) don't forget to add netty-tcnative .jar again
- Check that you have enabled OpenSSL in the configuration
- `searchguard.ssl.transport.enable_openssl_if_available: true`
- `searchguard.ssl.http.enable_openssl_if_available: true`

If you did all the steps above and start your nodes, you should see an entry similar to this in the logfile:

```
[INFO ][com.floragunn.searchguard.ssl.SearchGuardKeyStore] Open SSL OpenSSL 1.0.2d 9
Jul 2015 available
[INFO ][com.floragunn.searchguard.ssl.SearchGuardKeyStore] Open SSL available ciphers
[ECDHE-RSA-AES256-GCM-SHA384,...
```

If you face one of the following messages OpenSSL is not available and Search Guard SSL will use the built-in Java SSL implementation:

java.lang.ClassNotFoundException: org.apache.tomcat.jni.SSL

- netty-tcnative jar is missing, see above
- make sure you use the netty-tcnative jar **matching your platform**, either `_linux-x86.jar_` or `_64-fedora.jar_` or `_osx-x86_64.jar_` or `_windows-x86_64.jar_`

java.lang.UnsatisfiedLinkError

- OpenSSL is not installed, see above
- Apache Portable Runtime (APR) is not installed, see above

Further reading

- More about netty-tcnative can be found here:
- http://netty.io/wiki/forked-tomcat-native.html

Troubleshooting

Exception in thread "main" ElasticsearchException[No such keystore file ...]

```
Exception in thread "main" ElasticsearchException[No such keystore file
/Users/jkressin/Development/elasticsearch-2.1.0/config/node-01-keystore.jks]
    at
com.floragunn.searchguard.ssl.SearchGuardKeyStore.initSSLConfig(SearchGuardKeyStore.jav
a:172)
    at com.floragunn.searchguard.ssl.SearchGuardKeyStore.<init>
(SearchGuardKeyStore.java:129)
    at com.floragunn.searchguard.ssl.SearchGuardSSLModule.<init>
(SearchGuardSSLModule.java:29)
    at
com.floragunn.searchguard.ssl.SearchGuardSSLPlugin.nodeModules(SearchGuardSSLPlugin.jav
a:89)
...
```

The path to the keystore file is incorrect. Please check the settings for the configuration key

`searchguard.ssl.transport.keystore_filepath`

The value of this key is the path to the keystore file, **relative to the config directory**. For example, if you define `node-01-keystore.jks`, SG SSL expects the following file to exist:

`<es installation directory>/config/node-01-keystore.jks`

Exception in thread "main" ElasticsearchException[No such truststore file ...]

```
Exception in thread "main" ElasticsearchException[No such truststore file
/Users/jkressin/Development/elasticsearch-2.1.0/config/truststorea.jks]
    at
com.floragunn.searchguard.ssl.SearchGuardKeyStore.initSSLConfig(SearchGuardKeyStore.jav
a:182)
    at com.floragunn.searchguard.ssl.SearchGuardKeyStore.<init>
(SearchGuardKeyStore.java:129)
    at com.floragunn.searchguard.ssl.SearchGuardSSLModule.<init>
(SearchGuardSSLModule.java:29)
    at
com.floragunn.searchguard.ssl.SearchGuardSSLPlugin.nodeModules(SearchGuardSSLPlugin.jav
a:89)
```

The path to the truststore file is incorrect. Please check the settings for the configuration key

`searchguard.ssl.transport.truststore_filepath`

The value of this key is the path to the truststore file, **relative to the config directory**. For example, if you define `truststore.jks`, SG SSL expects the following file to exist:

```
<es installation directory>/config/truststore.jks
```

java.io.IOException: DerInputStream.getLength(): lengthTag=..., too big.

```
Exception in thread "main" ElasticsearchException[DerInputStream.getLength():
lengthTag=109, too big.]; nested: IOException[DerInputStream.getLength():
lengthTag=109, too big.];
Likely root cause: java.io.IOException: DerInputStream.getLength(): lengthTag=109, too
big.
    at sun.security.util.DerInputStream.getLength(DerInputStream.java:561)
    at sun.security.util.DerValue.init(DerValue.java:365)
    at sun.security.util.DerValue.<init>(DerValue.java:320)
    at sun.security.pkcs12.PKCS12KeyStore.engineLoad(PKCS12KeyStore.java:1872)
    at java.security.KeyStore.load(KeyStore.java:1433)
    at
com.floragunn.searchguard.ssl.SearchGuardKeyStore.initSSLConfig(SearchGuardKeyStore.jav
a:192)
    at com.floragunn.searchguard.ssl.SearchGuardKeyStore.<init>
(SearchGuardKeyStore.java:129)
    at com.floragunn.searchguard.ssl.SearchGuardSSLModule.<init>
(SearchGuardSSLModule.java:29)
```

SG SSL supports keystore/truststore files in either JKS or PKCS12 format. This exception means that the format you specified in the confoguration does not match the actual format of your keystore/truststore file. Please check the following configuration keys

- `searchguard.ssl.transport.keystore_type`
- `searchguard.ssl.transport.truststore_type`

Make sure it matches your keystore/truststore file format.

java.security.UnrecoverableKeyException: Password verification failed

```
xception in thread "main" ElasticsearchException[Keystore was tampered with, or
password was incorrect]; nested: IOException[Keystore was tampered with, or password
was incorrect]; nested: UnrecoverableKeyException[Password verification failed];
Likely root cause: java.security.UnrecoverableKeyException: Password verification
failed
    at sun.security.provider.JavaKeyStore.engineLoad(JavaKeyStore.java:770)
    at sun.security.provider.JavaKeyStore$JKS.engineLoad(JavaKeyStore.java:55)
    at java.security.KeyStore.load(KeyStore.java:1433)
    at
com.floragunn.searchguard.ssl.SearchGuardKeyStore.initSSLConfig(SearchGuardKeyStore.jav
a:192)
    at com.floragunn.searchguard.ssl.SearchGuardKeyStore.<init>
(SearchGuardKeyStore.java:129)
    at com.floragunn.searchguard.ssl.SearchGuardSSLModule.<init>
(SearchGuardSSLModule.java:29)
```

This simply means that the password for the keystore and/or truststore you provided in the configuration does not match the actual passord of your keystore/truststore. Check the following configuration entries to make sure the passwords are correct:

- `searchguard.ssl.transport.keystore_password`
- `searchguard.ssl.transport.truststore_password`

java.security.cert.CertificateException: No subject alternative names matching ... found

```
Caused by: java.security.cert.CertificateException: No subject alternative names
matching IP address ... found
    at sun.security.util.HostnameChecker.matchIP(HostnameChecker.java:154)
    at sun.security.util.HostnameChecker.match(HostnameChecker.java:91)
    at
sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:455)
    at
sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:436)
    at
sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:252)
    at
sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:136)
    at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1465)
```

This error is caused by a missing or invalid subject alternatice name (SAN) entry in the certificate, and only shows if hostname verification is enabled in the configuration (which is the default):

- `searchguard.ssl.transport.enforce_hostname_verification=true`

If you enable hostname verification, make sure that the SAN entry in your certificate exists, and that it matches the host portion of the URL used to make the request. If you used the `example.sh` script to generate your certificates, the SAN already contains `127.0.0.1`. If you still get the error mentioned above, make sure that Elasticsearch is bound to that IP address in your `elasticsearch.yml` file:

- `network.host: 127.0.0.1`

Alternatively, you can disable hostname verification:

- `searchguard.ssl.transport.enforce_hostname_verification=false`

Note: by disabling hostname verification you expose your cluster to man-in-th-middle attacks. Please fix your certificates settings rather than disabling hostname verification

java.security.cert.CertificateException: No subject alternative DNS name matching ... found.

```
Caused by: java.security.cert.CertificateException: No subject alternative DNS name
matching ... found.
    at sun.security.util.HostnameChecker.matchDNS(HostnameChecker.java:191)
    at sun.security.util.HostnameChecker.match(HostnameChecker.java:93)
    at
sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:455)
    at
sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:436)
    at
sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:252)
    at
sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:136)
    at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1465)
```

If you enabled hostname verification in the configuration:

- `searchguard.ssl.transport.enforce_hostname_verification=true`

and also enabled resolving the hostname against DNS:

- `searchguard.ssl.transport.resolve_hostname: true`

You must make sure that the hostname specified in the SAN field of your certificate can actually be resolved against DNS.

Alternatively, you can disable the DNS lookup:

- `searchguard.ssl.transport.resolve_hostname: false`