

Enabling TLS



- [Certificate rotation](#)
- [TLS-first Handshake](#)
- [TLS Timeout](#)
- [TLS Connection Rate Limit](#)
- [Certificate Authorities](#)
- [Self Signed Certificates for Testing](#)
 - [Problems With Self Signed Certificates](#)
 - [Client advertise not matching TLS names](#)
 - [Missing in Relevant Trust Stores](#)
 - [Missing Subject Alternative Name](#)
 - [Wrong Key Usage](#)
 - [Creating Self Signed Certificates for Testing](#)
- [TLS-Terminating Reverse Proxies](#)
 - [nats.js](#)
 - [nats.rs](#)
 - [nats.net \(v1\)](#)

The NATS server uses modern TLS semantics to encrypt client, route, and monitoring connections. [Check here for pitfalls.](#)

Server configuration revolves around a `tls` map, which has the following properties:

Property	Description		
<code>cert_file</code>	TLS certificate file.		
<code>key_file</code>	TLS certificate key file.		
<code>ca_file</code>	TLS certificate authority file . When not present, default to the system trust store.		

Property	Description		
<code>cipher_suites</code>	When set, only the specified TLS cipher suites will be allowed. Values must match the goLang version used to build the server.		
<code>curve_preferences</code>	List of TLS cipher curves to use in order.		
<code>insecure</code>	Skip certificate verification. This only applies to outgoing connections, NOT incoming client connections. NOT Recommended		
<code>min_version</code>	Minimum TLS version. Default is <code>"1.2"</code> .		
<code>timeout</code>	TLS handshake timeout in fractional seconds. Default set to <code>2</code> seconds.		
<code>verify</code>	If <code>true</code> , require and verify client certificates. To support use by Browser, this option does not apply to monitoring.		
<code>verify_and_map</code>	If <code>true</code> , require and verify client certificates and map certificate values for authentication purposes. Does not apply to monitoring either.		

Property	Description		
<code>verify_cert_and_check_known_urls</code>	<p>Only settable in a non client context where <code>verify:</code> <code>true</code> is the default (cluster/gateway). The incoming connections certificate's X509v3 Subject Alternative Name <code>DNS</code> entries will be matched against all urls in the configuration context that contains this tls map. If a match is found, the connection is accepted and rejected otherwise.</p> <p>Meaning for gateways we will match all DNS entries in the certificate against all gateway urls. For cluster, we will match against all route urls. As a consequence of this, dynamic cluster growth may require config changes in other clusters where this flag is true. DNS name checking is performed according to rfc6125. Only the full wildcard <code>*</code> is supported for the left most label. This would be one way to keep cluster growth flexible.</p>		
<code>pinned_certs</code>	<p>List of hex-encoded SHA256 of DER encoded public key fingerprints. When present, during the TLS handshake, the provided certificate's fingerprint is required to be present in the list or the connection is closed. This sequence of commands generates an entry for a provided certificate:</p> <pre>`openssl x509 -noout -pubkey -in</pre>	<code>openssl pkey -pubin -outform DER</code>	<code>openssl dgst -sha256`.</code>
<code>connection_rate_limit</code>	<p>Limit on the number of new TLS client connections accepted per second on the client listener. See TLS Connection Rate Limit. Default <code>0</code> disables the limit.</p>		

The simplest configuration:

```
tls: {  
  cert_file: "./server-cert.pem"  
  key_file:  "./server-key.pem"  
}
```

Or by using [server options](#):

```
nats-server --tls --tlscert=./server-cert.pem --tlskey=./server-key.pem
```

```
[21417] 2019/05/16 11:21:19.801539 [INF] Starting nats-server version 2.0.0  
[21417] 2019/05/16 11:21:19.801621 [INF] Git commit [not set]  
[21417] 2019/05/16 11:21:19.801777 [INF] Listening for client connections on 0.0.0.0:4222  
[21417] 2019/05/16 11:21:19.801782 [INF] TLS required for client connections  
[21417] 2019/05/16 11:21:19.801785 [INF] Server id is  
ND6ZZDQDQDKYQGDD6QN2Y26YEGLT6BMM0JZ2XJB2VASPVII3XD6RF0Q  
[21417] 2019/05/16 11:21:19.801787 [INF] Server is ready
```

Notice that the log indicates that the client connections will be required to use TLS. If you run the server in Debug mode with `-D` or `-DV`, the logs will show the cipher suite selection for each connected client:

```
[22242] 2019/05/16 11:22:20.216322 [DBG] 127.0.0.1:51383 - cid:1 - Client connection created  
[22242] 2019/05/16 11:22:20.216539 [DBG] 127.0.0.1:51383 - cid:1 - Starting TLS client  
connection handshake  
[22242] 2019/05/16 11:22:20.367275 [DBG] 127.0.0.1:51383 - cid:1 - TLS handshake complete  
[22242] 2019/05/16 11:22:20.367291 [DBG] 127.0.0.1:51383 - cid:1 - TLS version 1.2, cipher  
suite TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
```

When a `tls` section is specified at the root of the configuration, it also affects the monitoring port if `https_port` option is specified. Other sections such as `cluster` can specify a `tls` block.

Certificate rotation

Bear in mind that TLS keys and certificates for multiple clusters, or servers in different locations, rarely rotate at the exact same time and that Certificate Authorities do roll between multiple Intermediate certificates. Make sure that the `ca_file` contains both old and new root or intermediate CAs.

To apply changes to certificate files (even when only changing the content of those files) the server configuration needs to be reloaded. [Configuration reloading](#)

TLS-first Handshake

As of NATS v2.10.4

Client connections follow the model where, when a TCP connection is created to the server, the server will immediately send an [INFO protocol message](#) in clear text. This INFO protocol provides metadata, including whether the server requires a secure connection.

Some environments prefer having clients' TLS connections be initiated right away, that is, not having any traffic sent in clear text. It was possible to by-pass this using a websocket connection. However, if a websocket connection is not desired, the server can be configured to perform a TLS handshake before sending the INFO protocol message.

Only clients that implement an equivalent option would be able to connect if the server runs with this option enabled.

The configuration would look something like this:

```
tls: {  
  cert_file: "./server-cert.pem"  
  key_file:  "./server-key.pem"  
  handshake_first: true  
}
```

However, the parameter can be set to `auto` or a [Golang time duration](#) (e.g. `250ms`) to fallback to the original behavior. This is intended for deployments where it is known that not all clients have been upgraded to a client library providing the TLS-first handshake option.

After the delay has elapsed without receiving the TLS handshake from the client, the server reverts to sending the INFO protocol so that older clients can connect. Clients that do connect with the "TLS first" option will be marked as such in the monitoring's `Connz` page/result. It will allow the administrator to keep track of applications still needing to upgrade.

The configuration would be similar to:

```
tls: {  
  cert_file: "./server-cert.pem"  
  key_file:  "./server-key.pem"
```

```
handshake_first: auto
}
```

With the above value, the fallback delay used by the server is 50 milliseconds.

The duration can be explicitly set, say 300 milliseconds:

```
tls {
  cert_file: ...
  key_file: ...

  handshake_first: "300ms"
}
```

It is understood that any configuration other than "true" will result in the server sending the INFO protocol after the elapsed amount of time without the client initiating the TLS handshake. Therefore, for administrators who do not want any data transmitted in plain text, the value must be set to "true" only. It will require applications to be updated to a library that provides the option, which may or may not be readily available.

TLS Timeout

The `timeout` setting enables you to control the amount of time that a client is allowed to upgrade its connection to tls. If your clients are experiencing disconnects during TLS handshake, you'll want to increase the value, however, if you do be aware that an extended `timeout` exposes your server to attacks where a client doesn't upgrade to TLS and thus consumes resources. Conversely, if you reduce the TLS `timeout` too much, you are likely to experience handshake errors.

```
tls: {
  cert_file: "./server-cert.pem"
  key_file: "./server-key.pem"
  # clients will fail to connect (value is too low)
  timeout: 0.0001
}
```

TLS Connection Rate Limit

As of NATS v2.7.0

TLS handshakes are expensive, and a burst of reconnecting clients can keep the server busy negotiating new sessions instead of serving traffic. The `connection_rate_limit` setting caps the number of new TLS client connections the server will accept in any one second window. Connections above the cap are closed and the client receives the error `Connection throttling is active. Please try again later.`. The count of rejected connections is logged once per second so it is visible to operators.

```
tls: {  
  cert_file: "./server-cert.pem"  
  key_file:  "./server-key.pem"  
  connection_rate_limit: 1000  
}
```

The limit is a single counter for the whole server, not per-client or per-account. It only applies to the client listener, so cluster, gateway, leaf node, websocket and MQTT listeners are not affected by it. A value of `0`, the default, disables the limit.

Certificate Authorities

The `ca_file` file should contain one or more Certificate Authorities in PEM format, in a bundle. This is a common format.

When a certificate is issued, it is often accompanied by a copy of the intermediate certificate used to issue it. This is useful for validating that certificate. It is not necessarily a good choice as the only CA suitable for use in verifying other certificates a server may see.

Do consider though that organizations issuing certificates will change the intermediate they use. For instance, a CA might issue intermediates in pairs, with an active and a standby, and reserve the right to switch to the standby without notice. You probably would want to trust *both* of those for the `ca_file` directive, to be prepared for such a day, and then after the first CA has been compromised you can remove it. This way the roll from one CA to another will not break your NATS server deployment.

Self Signed Certificates for Testing

Explaining [Public key infrastructure](#), [Certificate Authorities \(CA\)](#) and [x509 certificates](#) fall well outside the scope of this document. So does an explanation on how to obtain a properly trusted certificates.

If anybody outside your organization needs to connect, get certs from a public certificate authority. Think carefully about revocation and cycling times, as well as automation, when picking a CA. If arbitrary applications inside your organization need to connect, use a cert from your in-house CA. If only resources inside a specific environment need to connect, that environment might have its own dedicated automatic CA, eg in Kubernetes clusters, so use that.

Only for **testing** purposes does it make sense to generate self-signed certificates, even your own CA. This is a **short** guide on how to do just that and what to watch out for.

“ DO NOT USE these certificates in production!!!

Problems With Self Signed Certificates

The issues and pitfalls listed here are not limited to self-signed certificates. You will most likely encounter them first in DEV environments when using those.

Client advertise not matching TLS names

NATS cluster advertises `host:port` of all nodes in a cluster to the connecting client. When connecting to a server via TLS the server name (or IP) is validated against the certificate presented by the server.

When using TLS, it is important to control the hostname that clients will use when discovering the server. By default, the cluster will advertise an IP address, which may result in a failed TLS hostname verification with an IP SANs error.

Set `advertise` or `cluster_advertise` in the cluster section to advertise verifiable server names. See [cluster_config.md](#)

Missing in Relevant Trust Stores

As they should, these are **not trusted** by the system your server or clients are running on.

One option is to specify the CA in every client you are using. In case you make use of `verify`, `verify_and_map` or `verify_cert_and_check_known_urls` you need to specify `ca_file` in the server. If you are having a more complex setup involving cluster, gateways or leaf nodes, `ca_file` needs to be present in `tls` maps used to connect to the server with self-signed certificates. While this works for server and libraries from the NATS ecosystem, you will experience issues when connecting with other tools such as your Browser.

Another option is to configure your system's trust store to include self-signed certificate(s). Which trust store needs to be configured depends on what you are testing.

- This may be your OS for server and certain clients.
- The runtime environment for other clients like Java, Python or Node.js.
- Your browser for monitoring endpoints and websockets.

Please check your system's documentation on how to trust a particular self-signed certificate.

Missing Subject Alternative Name

Another common problem is failed [identity validation](#). The IP or DNS name to connect to needs to match a [Subject Alternative Name \(SAN\)](#) inside the certificate. Meaning, if a client/browser/server connect via tls to `127.0.0.1`, the server needs to present a certificate with a SAN containing the IP `127.0.0.1` or the connection will be closed with a handshake error.

When `verify_cert_and_check_known_urls` is specified, [Subject Alternative Name \(SAN\)](#) `DNS` records are necessary. In order to successfully connect there must be an overlap between the `DNS` records provided as part of the certificate and the urls configured. If you dynamically grow your cluster and use a new certificate, this route or gateway the server connects to will have to be reconfigured to include an url for the new server. Only then can the new server connect. If the `DNS` record is a wildcard, matching according to [rfc6125](#) will be performed. Using certificates with a wildcard [Subject Alternative Name \(SAN\)](#) and configuration with url(s) that would match are a way to keep the flexibility of dynamic cluster growth without configuration changes in other clusters.

Wrong Key Usage

When generating your certificate you need to make sure to include the right purpose for which you want to use the certificate. This is encoded in [key usage](#) and [extended key usage](#). The necessary values for key usage depend on the ciphers used. `Digital Signature` and `Key Encipherment` are an interoperable choice.

With respect to NATS the relevant values for extended key usage are:

- `TLS WWW server authentication` - To authenticate as server for incoming connections. A NATS server will need a certificate containing this.
- `TLS WWW client authentication` - To authenticate as client for outgoing connections. Only needed when connecting to a server where `verify`, `verify_and_map` or `verify_cert_and_check_known_urls` are specified. In these cases, a NATS client will need a certificate with this value.
 - [Leaf node](#) connections can be configured with `verify` as well. Then the connecting NATS server will have to present a certificate with this value too. Certificates containing both values are an option.
 - [Cluster](#) connections always have `verify` enabled. Which server acts as client and server comes down to timing and therefore can't be individually configured. Certificates containing both values are a must.

- [Gateway](#) connections always have `verify` enabled. Unlike cluster outgoing connections can specify a separate cert. Certificates containing both values are an option that reduce configuration.

Note that it's common practice for non-web protocols to use the `TLS WWW` authentication fields, as a matter of history those have become embedded as generic options.

Creating Self Signed Certificates for Testing

The simplest way to generate a CA as well as client and server certificates is [mkcert](#). This zero config tool generates and installs the CA into your **local** system trust store(s) and makes providing SAN straight forward. Check its [documentation](#) for installation and your system's trust store. Here is a simple example:

Generate a CA as well as a certificate, valid for server authentication by `localhost` and the IP `::1` (`-cert-file` and `-key-file` overwrite default file names). Then start a NATS server using the generated certificate.

```
mkcert -install
mkcert -cert-file server-cert.pem -key-file server-key.pem localhost ::1
nats-server --tls --tlscert=server-cert.pem --tlskey=server-key.pem -ms 8222
```

Now you should be able to access the monitoring endpoint `https://localhost:8222` with your browser. `https://127.0.0.1:8222` however should result in an error as `127.0.0.1` is not listed as SAN. You will not be able to establish a connection from another computer either. For that to work you have to provide appropriate DNS and/or IP [SAN\(s\)](#)

To generate certificates that work with `verify` and `cluster`/`gateway`/`leaf_nodes` provide the `-client` option. It will cause the appropriate key usage for client authentication to be added. This example also adds a SAN email for usage as user name in `verify_and_map`.

```
mkcert -client -cert-file client-cert.pem -key-file client-key.pem localhost ::1
email@localhost
```

“ Please note:

- That client refers to connecting process, not necessarily a NATS client.

- `mkcert -client` will generate a certificate with key usage suitable for client **and** server authentication.

Examples in this document make use of the certificates generated so far. To simplify examples using the CA certificate, copy `rootCA.pem` into the same folder where the certificates were generated. To obtain the CA certificate's location use this command:

```
mkcert -CAROOT
```

Once you are done testing, remove the CA from your **local** system trust store(s).

```
mkcert -uninstall
```

Alternatively, you can also use [openssl](#) to [generate certificates](#). This tool allows a lot more customization of the generated certificates. It is **more complex** and does **not manage** installation into the system trust store(s).

However, for inspecting certificates it is quite handy. To inspect the certificates from the above example execute these commands:

```
openssl x509 -noout -text -in server-cert.pem
openssl x509 -noout -text -in client-cert.pem
```

TLS-Terminating Reverse Proxies

Using a [TLS-terminating reverse proxy](#) with NATS requires some specific configuration on the server. In a typical proxy scenario, the client to proxy communication is secured and the proxy to server is insecure. This causes a "mismatch" because the server appears to be insecure but the client is told to connect securely. To fix this, the server must be configured as "tls available". This is done via an empty `tls` block and the `allow_non_tls` flag.

```
tls {}
allow_non_tls: true
```

Once this is configured, your client can connect to the proxy with normal (language specific) tls configuration. Please make sure you are using the appropriate version of your language specific client.

Client	Version
--------	---------

nats.go	v1.31.0
nats.js	2024.1.2
nats.java	2.18.0
nats.rs	0.33
nats.net.v2	2.0.0
nats.net (v1)	1.1.5

nats.js

See: <https://github.com/nats-io/nats.js/issues/369>

nats.rs

See: <https://github.com/nats-io/nats.rs/blob/main/async-nats/src/connector.rs>

nats.net (v1)

See: <https://github.com/nats-io/nats.net.v1/tree/main/src/Samples/TlsVariationsExample>

- #3
- cocobook-sync

2026-10-28 11:13:56 UTC
- cocobook-sync

2026-10-29 01:42:38 UTC