DigiTalk

Mastering SSL/TLS, PKI and Certificate Management

A Complete Beginner-to-Practitioner Guide

14.1.2 & 15.1.1
DT246055-03
July 2026

DigiTalk

Table of Contents

Module 1 - Introduction to SSL/TLS

What is SSL?

SSL (Secure Sockets Layer) is a protocol that was designed to let two computers — usually a browser and a server — talk to each other over the internet without anyone else being able to read or tamper with the conversation. Think of it as a locked, sealed envelope wrapped around your data before it travels across a public network.

Why SSL was replaced by TLS

SSL had serious security weaknesses (POODLE, weak ciphers, broken handshake design). Instead of continuing to patch an aging protocol, the industry created TLS (Transport Layer Security) as its successor. TLS is really "SSL, fixed and modernized" — the name changed, but the purpose stayed the same.

- SSL 2.0 and SSL 3.0 — obsolete, insecure, must not be used.
- TLS 1.0 / 1.1 — improved on SSL, but now considered outdated.
- TLS 1.2 / 1.3 — the current, secure standards used today.

Note: People still say "SSL certificate" out of habit — in practice, almost all modern connections actually use TLS, not SSL.

SSL vs TLS

Aspect	SSL	TLS
Status	Deprecated / insecure	Current standard
Versions	SSL 2.0, SSL 3.0	TLS 1.0 – 1.3
Handshake	Weaker, fewer safeguards	Stronger, faster (esp. TLS 1.3)
Everyday usage	Should never be enabled	Used by virtually all HTTPS traffic

Why Encryption Is Needed

Without encryption, data travels across the internet as plain, readable text — anyone sitting on the network path (a shared Wi-Fi hotspot, a compromised router, an ISP) can read or alter it. Encryption exists to guarantee four things:

Confidentiality

Only the intended recipient can read the data. Example: your password, typed on a login page, is scrambled so an eavesdropper on public Wi-Fi only sees meaningless characters.

Integrity

The data cannot be silently changed in transit without detection. Example: if someone tries to alter the amount in an online bank transfer while it's travelling over the network, the tampering is detected and the connection is rejected.

Authentication

You are actually talking to who you think you are talking to. Example: TLS certificates prove that "bank.com" is really operated by the bank, not by an impersonator running a look-alike site.

Non-repudiation

The sender cannot later deny having sent the message, because their digital signature is mathematically tied to them. Example: a digitally signed contract or a code-signed software update proves who produced it.

Real-World HTTPS Communication

HTTPS is simply HTTP running on top of TLS. When you type an address starting with https://, your browser first performs a TLS handshake with the server (agreeing on encryption and verifying the server's certificate), and only after that succeeds does the actual web page request/response travel — now encrypted — over that secured channel.

SSL Handshake Overview

The handshake is the negotiation phase that happens before any real data is exchanged. In simple terms:

- The client says hello and lists what encryption it supports.
- The server replies with its certificate (proof of identity) and its choice of encryption.
- Both sides use public/private key math to agree on a shared secret key.
- From that point on, all traffic is encrypted using that shared secret.

Public Key vs Private Key

These two keys are mathematically linked but serve opposite purposes:

- Public Key — can be shared with anyone; used to encrypt data or verify a signature.
- Private Key — kept secret by the owner; used to decrypt data or create a signature.

Example: anyone can use your public key to lock (encrypt) a message meant only for you, but only your private key can unlock (decrypt) it.

Symmetric vs Asymmetric Encryption

Aspect	Symmetric Encryption	Asymmetric Encryption
Keys used	One shared secret key	A public/private key pair
Speed	Very fast	Much slower
Used for	Encrypting the actual bulk traffic	The handshake / key exchange, signatures
Example algorithm	AES	RSA, ECDSA

In practice, TLS uses both: asymmetric encryption to safely agree on a secret during the handshake, then fast symmetric encryption (like AES) to protect the actual data that follows — the best of both worlds.

DigiTalk

Hashing

A hash function takes any input and produces a fixed-length "fingerprint" of that data. The same input always produces the same hash, but even a tiny change in the input produces a completely different hash. Hashing is one-way — you cannot reverse a hash back into the original data. It is used to verify integrity: if a file's hash matches the expected value, the file was not altered.

```
Input:  "Hello World"
SHA-256 hash: a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146

Input:  "Hello World!"   (just one character added)
SHA-256 hash: 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069
```

Digital Signature

A digital signature proves both integrity and authenticity. The sender hashes the data, then encrypts that hash with their private key. Anyone can then use the sender's public key to decrypt the signature and compare it to a freshly computed hash of the data — if they match, the data is genuine and untampered.

Module 2 - PKI (Public Key Infrastructure)

PKI is the overall system of people, policies, and technology that makes public-key cryptography trustworthy on a large scale. Most introductory courses skip this, but it is the foundation everything else in this document builds on: without PKI, there would be no reliable way to know that a public key really belongs to the party it claims to belong to.

PKI Introduction

PKI answers one core question: "How do I trust a stranger's public key?" The answer is a chain of digital certificates, each vouching for the next, ultimately anchored to a small set of universally trusted authorities.

Certificate Authority (CA)

A Certificate Authority is a trusted organization that verifies identities and then issues digitally signed certificates confirming "this public key really belongs to this entity." Examples of well-known public CAs include DigiCert, Sectigo, and Let's Encrypt.

Root CA

The top of the trust chain. A Root CA's certificate is self-signed and is pre-installed as trusted in operating systems and browsers. Because root keys are extremely sensitive, they are kept offline and used as rarely as possible.

Intermediate CA

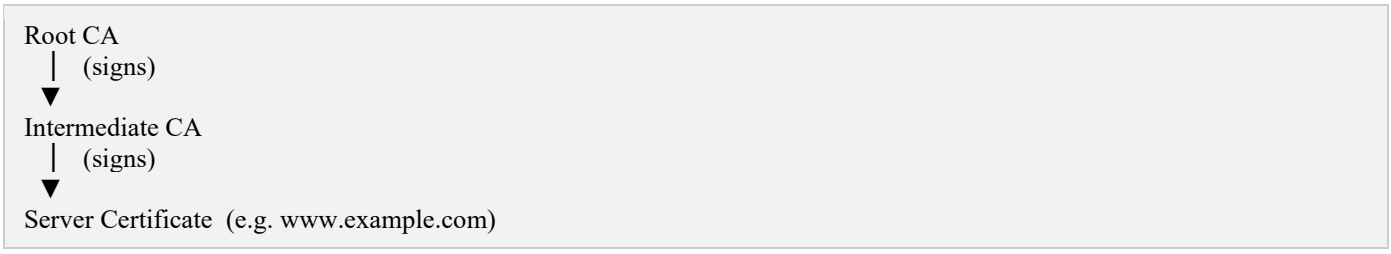
A Root CA rarely signs end-user certificates directly. Instead, it signs one or more Intermediate CAs, which then do the day-to-day work of issuing certificates. This limits exposure of the root's private key — if an intermediate is ever compromised, it can be revoked without invalidating the root.

End Entity Certificate

Also called a "leaf" or "server" certificate — this is the actual certificate installed on a website or application, issued and signed by an intermediate CA.

Certificate Chain / Chain of Trust

A browser trusts an end-entity certificate by walking up the chain: the server certificate is signed by an intermediate, which is signed by the root, which the browser already trusts. If every link in that chain verifies correctly, the certificate is accepted.



A certificate signed by its own private key rather than by a CA. It provides encryption but no independently verifiable identity — nobody outside the creator vouches for it. Useful for internal testing, but browsers show warnings because there is no chain of trust.

CA-Signed Certificate

A certificate issued and signed by a recognized CA. Because the CA's root (or an intermediate under it) is already trusted by browsers/operating systems, the certificate is automatically trusted without any warning.

Why Browsers Trust DigiCert but Not Self-Signed Certificates

Operating systems and browsers ship with a curated list of trusted Root CA certificates (the "trust store"). DigiCert's root is on that list because DigiCert underwent rigorous audits and agreed to strict issuance rules. A self-signed certificate isn't on that list, and there's no audited third party vouching for it — so the browser has no basis to trust it automatically, and shows a warning instead.

Module 3 - Keys

This module explains the public/private key pair from scratch — what each key is used for and how they work together.

Public Key

The public key can be freely shared with anyone in the world. It is used for:

- Encryption — anyone can use it to encrypt a message that only the matching private key can decrypt.
- Signature Verification — anyone can use it to check that a signature was really created by the matching private key.

Private Key

The private key must be kept secret, known only to its owner. It is used for:

- Decryption — decrypting data that was encrypted with the matching public key.
- Digital Signature — signing data to prove authenticity; anyone can verify it using the public key.

Example Flow: Encrypting Data With a Public Key

```
    Public Key
        ↓
[ Encrypt Data ]
        ↓
    Internet
        ↓
    Private Key
[ Decrypt Data ]
```

In this flow, anyone holding the public key can lock the data, but only the one person holding the matching private key can unlock it — this is exactly how a client encrypts session information toward a server during parts of the TLS handshake.

DigiTalk

Module 4 - Certificate

A digital certificate is essentially an electronic ID card. It binds a public key to an identity (a person, server, or organization) and is digitally signed by a CA to prove that binding is genuine.

What Is Inside a Certificate?

Every X.509 certificate (the standard format used for TLS) contains a defined set of fields:

Subject
The identity the certificate belongs to — for example, the website's domain name (Common Name) and organization details.
Issuer
The CA that signed and issued the certificate — this points to the next link up the chain of trust.
Public Key
The subject's public key, which the certificate is vouching for.
Validity
The "Not Before" and "Not After" dates that define the window during which the certificate is considered valid.
Serial Number
A unique number assigned by the CA to this certificate — used to identify or revoke it individually.
Signature Algorithm
The algorithm the CA used to sign the certificate, e.g. SHA256withRSA.
Extensions
Additional fields that add extra rules and metadata to the certificate:

- SAN (Subject Alternative Name) — the list of domain names/IP addresses the certificate is actually valid for. Modern browsers rely on this instead of the Common Name.
- Key Usage — restricts what the key may cryptographically be used for (e.g. digital signature, key encipherment).
- Extended Key Usage — further restricts the purpose, e.g. "server authentication" or "client authentication".

Viewing a Certificate With OpenSSL

```
openssl x509 -in certificate.crt -text -noout
```

Running this prints every field described above in human-readable form. A trimmed example of what you'd see:

```
Certificate:
    Data:
        Serial Number: 04:3a:9f:1e:...
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=DigiCert Inc, CN=DigiCert TLS RSA SHA256 2020 CA1
        Validity
            Not Before: Jan 10 00:00:00 2026 GMT
            Not After : Jan 10 23:59:59 2027 GMT
        Subject: CN=www.example.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                DNS:www.example.com, DNS:example.com
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
```

Module 5 - Keystore

A keystore is a protected file that stores cryptographic keys and certificates. This is one of the most important modules for anyone doing hands-on TLS/Java work, because almost every real-world configuration task revolves around keystore files.

What Is a Keystore?

A keystore is a secure, password-protected container file that holds one or more private keys, their matching public certificates, and optionally trusted CA certificates. Applications (like a Java web server) load a keystore at startup so they have their identity (private key + certificate) ready to present during a TLS handshake.

Keystore vs Truststore

Aspect	Keystore	Truststore
Holds	Your own private key + your certificate	Other parties' trusted (public) certificates
Purpose	Prove your own identity to others	Decide whom you trust / accept connections from
Analogy	Your passport and its private signing pen	The list of governments whose passports you accept

Note: Technically a keystore and truststore can be the exact same file format — the distinction is about what you choose to put in it and how the application is configured to use it.

JKS — Java KeyStore

- Oracle's own proprietary format.
- One of the oldest formats, specific to the Java ecosystem.
- Was the default keystore format before Java 9.

Advantages

- Deep, native integration with Java and tools like keytool.
- Long history — well documented in older Java projects.

Disadvantages

- Proprietary — not readable by OpenSSL or non-Java tools without conversion.
- Weaker legacy protection for private keys compared to PKCS12.
- No longer the recommended or default format.

PKCS12

- An industry-standard container format (not tied to Java).
- Supported virtually everywhere — browsers, OpenSSL, Java, .NET, mobile platforms.
- Recommended format for new work.
- Cross-platform by design.
- Fully OpenSSL compatible — easy to inspect, convert, and script.
- Has been the default keystore format since Java 9.

Comparison Table

Feature	JKS	PKCS12
Java Only	Yes	No
Industry Standard	No	Yes
OpenSSL Support	No	Yes
Default in Java 9+	No	Yes
Recommended	No	Yes

Recommendation

For any new environment being set up today, always use PKCS12 (.p12 / .pfx) rather than JKS. Only continue using JKS when maintaining an older system that already depends on it.

DigiTalk

Module 6 - Keytool

What Is Keytool?

keytool is a command-line utility bundled with the Java Development Kit (JDK) used to create, manage, and inspect keystores, truststores, and the certificates inside them.

Where It Is Located

```
JAVA_HOME/bin/keytool
```

Because it ships inside the JDK's bin directory, it is available on the command line anywhere JAVA_HOME/bin is on the system PATH.

Common Commands

Option	What It Does
-genkeypair	Generates a new public/private key pair and wraps it in a self-signed certificate.
-list	Lists the entries (aliases) contained in a keystore.
-importcert	Imports a certificate (e.g. a CA-signed cert or a trusted root) into a keystore.
-exportcert	Exports a certificate from a keystore into its own file.
-delete	Removes an entry (by alias) from a keystore.
-changealias	Renames the alias of an existing keystore entry.
-storepasswd	Changes the password protecting the entire keystore.
-keypasswd	Changes the password protecting an individual private key entry.
-certreq	Generates a Certificate Signing Request (CSR) from an existing key pair.

Example — listing everything inside a keystore:

```
keytool -list -v -keystore server.p12 -storetype PKCS12
```

DigiTalk

Module 7 - OpenSSL

Why OpenSSL Exists

OpenSSL is a free, open-source toolkit implementing SSL/TLS and general cryptography. Unlike keytool, it is not tied to Java — it is the universal, cross-platform tool used across Linux, web servers, embedded devices, and almost every non-Java stack for generating keys, certificates, and CSRs, and for testing TLS connections directly.

Keytool vs OpenSSL

Aspect	Keytool	OpenSSL
Origin	Bundled with Java (Oracle)	Open source, independent project
Native format	Keystore (JKS/PKCS12)	PEM / raw OpenSSL formats
Typical usage	Java applications	Linux servers, web servers, general purpose

Common OpenSSL Commands

Command	Purpose
openssl version	Shows the installed OpenSSL version.
openssl x509	Displays or manipulates X.509 certificates.
openssl rsa	Displays or manipulates RSA private keys.
openssl pkcs12	Creates or extracts data from a PKCS12 (.p12/.pfx) file.
openssl verify	Verifies a certificate chain against trusted CAs.
openssl s_client	Connects to a remote TLS server to inspect its certificate and handshake.

Example — checking a live website's certificate from the command line:

```
openssl s_client -connect www.example.com:443 -servername www.example.com
```

DigiTalk

Module 8 - Creating Self-Signed Certificates

This module walks through generating a self-signed key pair and certificate in a JKS/PKCS12 keystore using keytool, and explains every option along the way.

The Command

```
keytool -genkeypair \
  -alias myserver \
  -keyalg RSA \
  -keysize 2048 \
  -sigalg SHA256withRSA \
  -storetype PKCS12 \
  -keystore server.p12 \
  -storepass changeit \
  -keypass changeit \
  -validity 365 \
  -dname "CN=digitalk.local, OU=IT, O=Digitalk, L=Bengaluru, ST=Karnataka, C=IN" \
  -ext SAN=dns:digitalk.local,dns:www.digitalk.local
```

Explaining Every Option

- -alias — the nickname used to refer to this key/certificate entry inside the keystore (you'll use this name again when listing, exporting, or deleting it).
- -keyalg RSA — tells keytool which algorithm to use to generate the public/private key pair. RSA is the most common; ECDSA is a modern, smaller-key alternative.
- -keysize 2048 — the length of the key in bits (explained in detail below).
- -sigalg SHA256withRSA — the algorithm used to sign the certificate (explained below).
- -storetype — the keystore format to create (JKS or PKCS12).
- -keystore — the output file name for the keystore.
- -storepass — the password protecting the whole keystore file.
- -keypass — the password protecting this specific private key entry (often set the same as -storepass for PKCS12).
- -validity 365 — how many days the generated certificate remains valid.
- -dname — the Distinguished Name identifying the certificate subject (broken down below).
- -ext — additional X.509 extensions to embed, such as the SAN list.

-keyalg RSA — What It Means

RSA is the algorithm keytool uses to mathematically generate the linked public/private key pair. Specifying RSA tells keytool exactly which mathematical approach to use for that generation.

-keysize 2048 — Why 2048?

- 1024-bit — considered weak by modern standards; crackable with enough computing resources; should not be used.
- 2048-bit — the current baseline recommendation; strong enough for the foreseeable future and well supported everywhere.
- 4096-bit — even stronger, but noticeably slower to generate and use in every handshake, with limited real-world security benefit over 2048 for most use cases.
-

Key Size	Security	Performance	Recommendation
1024-bit	Weak / broken	Fastest	Do not use
2048-bit	Strong	Good balance	Recommended default
4096-bit	Very strong	Noticeably slower	Use for long-lived / high-value roots only

-sigalg SHA256withRSA — What It Means

This tells keytool to sign the certificate by first hashing its contents with SHA-256, then encrypting that hash using the RSA private key.

- SHA256 — the hashing algorithm that produces a fixed-size fingerprint of the certificate's data, so any tampering can be detected.
- RSA — the algorithm used to "sign" (encrypt) that fingerprint with the private key, so the signature can later be verified with the matching public key.

Combined together, SHA256withRSA means: hash it with SHA-256, then sign that hash with RSA — giving both integrity (via the hash) and authenticity (via the signature).

-dname — Breaking Down the Distinguished Name

Field	Meaning	Example
CN	Common Name — the primary identity, usually a hostname	digitalk.local
OU	Organizational Unit — the department/team	IT
O	Organization — the company/entity name	Digitalk
L	Locality — city	Bengaluru
ST	State/Province	Karnataka
C	Country (2-letter code)	IN

Explaining SAN

```
-ext SAN=dns:digitalk.local
```

SAN (Subject Alternative Name) is an extension listing every hostname (and optionally IP address) the certificate should be valid for. This is very important because modern browsers ignore the CN field entirely for hostname matching and check the SAN list instead.

CN vs SAN

Aspect	CN (Common Name)	SAN (Subject Alternative Name)
Historical role	Used to be the only hostname field checked	Introduced to allow multiple hostnames
Modern browser behavior	Ignored for hostname validation	The only field actually checked
Recommendation	Still set for readability/compatibility	Always include the real hostname(s) here

Note: A certificate without a matching SAN entry will fail hostname verification in every modern browser, even if the CN looks correct.

Module 9 - CSR (Certificate Signing Request)

What Is a CSR?

A CSR is a file you generate from your own key pair and send to a CA when you want a properly signed (trusted) certificate instead of a self-signed one. It essentially says: "Here is my public key and my identity details — please verify me and sign a certificate for it."

What Is Inside a CSR

- The applicant's public key.
- The Distinguished Name (CN, O, OU, L, ST, C) describing who is requesting the certificate.
- Requested extensions, such as the SAN list of domains.
- A signature created with the applicant's own private key, proving they genuinely hold that key.

Who Signs It

The CSR itself is signed by the requester (using their own private key, to prove key ownership) — but the resulting certificate is signed by the CA after they verify the requester's identity and domain ownership.

Why the CA Needs the CSR

The CSR gives the CA everything it needs to issue a certificate: your public key and your claimed identity. The CA never needs — and should never receive — your private key; it stays on your machine at all times.

Generating a CSR With Keytool

```
keytool -certreq -alias myserver -keystore server.p12 -storetype PKCS12 -storepass changeit -file myserver.csr
```

This produces a myserver.csr file, based on the key pair already stored under the myserver alias, which you then submit to your chosen CA.

DigiTalk

Module 10 - Importing Certificates

-importcert

Once a CA returns a signed certificate (or certificate chain), it must be imported back into the keystore, replacing the self-signed placeholder certificate that keytool originally created alongside the key pair.

What Gets Imported

- Root CA certificate — the top-level trust anchor.
- Intermediate certificate(s) — the middle links in the chain.
- Server Certificate — the actual certificate issued for your domain/application.

Import Order Matters

```
Root CA
  ↓ (import first)
Intermediate
  ↓ (import second)
Server Certificate
  ↓ (import last, into the same alias as the original key pair)
```

Importing out of order (or skipping the intermediate) is one of the most common causes of the "unable to find valid certification path" error covered later in the troubleshooting module.

Example Commands

```
keytool -importcert -alias root -file root-ca.crt -keystore server.p12 -storetype PKCS12 -storepass changeit
keytool -importcert -alias intermediate -file intermediate-ca.crt -keystore server.p12 -storetype PKCS12 -storepass changeit
keytool -importcert -alias myserver -file server.crt -keystore server.p12 -storetype PKCS12 -storepass changeit
```

Module 11 - Truststore

Keystore vs Truststore, Revisited

	Keystore	Truststore
Contains	Your own private key + your own certificate	Certificates you have decided to trust
Answers the question	"Who am I, and can I prove it?"	"Do I trust the other side's identity?"

cacerts — the Default Java Truststore

Every JDK installation ships with a built-in truststore file named cacerts, pre-loaded with the major public Root CA certificates (DigiCert, Sectigo, Let's Encrypt, etc). Java applications use this file by default to decide which server certificates to trust, unless a custom truststore is explicitly configured.

```
JAVA_HOME/lib/security/cacerts
```

The default password for the cacerts file is traditionally changeit — it should be changed in any production environment.

Example — listing the trusted CAs already inside it:

```
keytool -list -keystore "$JAVA_HOME/lib/security/cacerts" -storepass changeit
```

Module 12 - Certificate Formats

There are many file extensions floating around this space, and they cause a lot of confusion. This single module clears them all up.

Formats Explained

- .cer - a certificate file, usually DER (binary) encoded, sometimes PEM.
- .crt - a certificate file, commonly used on Linux/Unix systems; can be PEM or DER.
- .pem - Base64 (text) encoded data wrapped in ----BEGIN/END---- markers; can hold certificates, keys, or chains.
- .der - the raw binary encoding of a certificate (no Base64 text wrapper).
- .p7b - a PKCS#7 file that can bundle a certificate chain (no private key), commonly used with Windows/Java tooling.
- .pfx / .p12 - a PKCS#12 archive bundling a private key together with its certificate (and chain), usually password protected.
- .jks - a Java KeyStore file, holding one or more key/certificate entries.
- .key - typically a private key file, often PEM encoded.
- .csr - a Certificate Signing Request, the file sent to a CA to request a signed certificate.

Comparison Table

Extension	Typically Contains
.crt	Certificate
.cer	Certificate
.pem	Base64-encoded certificate, key, or chain
.der	Binary-encoded certificate
.p12	Private Key + Certificate (bundle)
.jks	Java Keystore (keys + certificates)

DigiTalk

Module 13 - Conversion

Different tools and platforms expect different formats, so converting between them is one of the most useful day-to-day skills in this whole subject.

Why Conversion Is Needed

A Java app might require a JKS or PKCS12 keystore, while an Nginx server expects PEM files, and a Windows server might want a PFX bundle. Rather than generating separate keys for each platform, the same key/certificate material is converted between formats as needed.

JKS → PKCS12

```
keytool -importkeystore -srckeystore server.jks -srcstoretype JKS -destkeystore server.p12 -deststoretype PKCS12
```

PKCS12 → JKS

```
keytool -importkeystore -srckeystore server.p12 -srcstoretype PKCS12 -destkeystore server.jks -deststoretype JKS
```

PEM → PKCS12

```
openssl pkcs12 -export -in cert.pem -inkey key.pem -out server.p12 -name myserver
```

PKCS12 → PEM

```
openssl pkcs12 -in server.p12 -out cert.pem -clcerts -nokeys
openssl pkcs12 -in server.p12 -out key.pem -nocerts -nodes
```

CRT + KEY → PKCS12

```
openssl pkcs12 -export -in server.crt -inkey server.key -out server.p12 -name myserver -certfile ca-chain.crt
```

DER → PEM

```
openssl x509 -inform der -in certificate.der -out certificate.pem
```

PEM → DER

```
openssl x509 -outform der -in certificate.pem -out certificate.der
```

DigiTalk

Module 14 - Viewing Certificates

A quick reference of the commands used to inspect certificates, keys, and keystores, whether working with keytool or OpenSSL.

keytool -list

Lists all aliases in a keystore, optionally with full detail.

```
keytool -list -v -keystore server.p12 -storetype PKCS12 -storepass changeit
```

keytool -printcert

Prints the details of a standalone certificate file (not one already inside a keystore).

```
keytool -printcert -file server.crt
```

openssl x509

Prints the full text contents of a PEM/DER certificate.

```
openssl x509 -in server.crt -text -noout
```

openssl pkcs12

Lists the contents (certificates and keys) inside a PKCS12 file.

```
openssl pkcs12 -info -in server.p12 -noout
```

openssl rsa

Displays details about an RSA private key, and can verify that a key file is valid.

```
openssl rsa -in server.key -check -noout
```

Module 15 - SSL/TLS Handshake (Detailed)

This module walks through the handshake step by step — the negotiation that happens every time a TLS connection is opened, before any application data flows.

Handshake Flow

```
Client                                  Server
  |------------------- Hello -------------------------->|
  |<------------------ Certificate ---------------------|
  |------------------- Key Exchange -------------------->|
  |<===== Finished (both sides) ========================>|
  |<==================== Encrypted Traffic ==============>|
```

Hello

The client sends a "Client Hello" listing the TLS versions and cipher suites it supports, plus a random value. The server responds with a "Server Hello", picking the TLS version and cipher suite to use, plus its own random value.

Certificate

The server sends its certificate (and usually its intermediate chain) to prove its identity. The client validates this chain against its trusted root store, checks the hostname against the SAN list, and checks the validity dates.

Key Exchange

Both sides use asymmetric cryptography to securely agree on a shared secret, without ever transmitting that secret in the clear. In TLS 1.3 this typically uses Diffie-Hellman key exchange; in older TLS versions RSA key exchange was also common.

Finished

Both the client and server send a "Finished" message, encrypted with the newly agreed keys, confirming that the handshake completed correctly and that both sides derived the same shared secret.

Encrypted Traffic

From this point forward, all application data (the actual HTTP request/response, etc.) is encrypted using fast symmetric encryption (like AES) with the shared secret negotiated during the handshake.

Module 16 - TLS Versions

Version	Status
SSL 2.0	Deprecated — insecure, must not be used
SSL 3.0	Deprecated — insecure, must not be used (POODLE vulnerability)
TLS 1.0	Deprecated — officially retired by major browsers/standards bodies
TLS 1.1	Deprecated — officially retired by major browsers/standards bodies
TLS 1.2	Current and widely supported — still secure when configured correctly
TLS 1.3	Current and recommended — faster handshake, stronger default security

The general rule for any production system today: support TLS 1.2 and TLS 1.3 only, and disable everything older.

DigiTalk

Module 17 - Cipher Suites

A cipher suite is the specific bundle of algorithms negotiated during the handshake to actually protect the connection. Breaking one apart shows exactly what it does.

Example: TLS_AES_256_GCM_SHA384

- TLS - indicates this is a TLS 1.3 cipher suite naming convention.
- AES - the symmetric encryption algorithm used to protect the bulk of the traffic.
- 256 - the key length in bits used by AES; larger means stronger encryption.
- GCM - Galois/Counter Mode, the mode of operation that also provides built-in integrity checking (detects tampering) alongside encryption.
- SHA384 - the hash algorithm used for the handshake's integrity/authentication (HMAC) purposes.

Put together, this cipher suite means: encrypt bulk data with 256-bit AES in GCM mode, and use SHA-384 for the handshake's integrity/authentication needs.

Module 18 - Mutual SSL

One-Way SSL

The standard, everyday setup: only the server presents a certificate, and the client verifies it. This is what happens on virtually every HTTPS website you visit — the server proves its identity, but you (the client) don't have to prove yours cryptographically.

Two-Way SSL (Mutual TLS)

Both the server and the client present certificates, and both sides verify each other. This is common in business-to-business integrations, banking, and internal microservice communication, where the server also needs cryptographic proof of exactly who is connecting to it.

Client Certificate

In mutual SSL, the client holds its own private key and certificate (issued by a CA the server trusts) and presents it during the handshake, just as the server presents its own certificate. The server then validates the client's certificate chain the same way a browser validates a server's.

WebLogic Configuration (Overview)

To enable mutual SSL on an Oracle WebLogic server, the server's SSL configuration must be set to request (or require) client certificates, and its trust keystore must contain the CA certificate(s) that signed the expected client certificates. This ties directly into the WebLogic-specific setup covered in the next module.

Module 19 - WebLogic SSL

Oracle WebLogic Server has its own terminology and configuration layer around TLS. This module maps the general concepts covered earlier onto WebLogic's specific setup.

Identity Keystore

WebLogic's term for a keystore — holds the server's own private key and certificate, used to prove WebLogic's identity to clients (equivalent to the general "keystore" concept).

Trust Keystore

WebLogic's term for a truststore — holds the CA certificates WebLogic will trust when verifying certificates presented by others (equivalent to the general "truststore" concept).

Demo Identity / Demo Trust

WebLogic ships with a built-in demonstration keystore and truststore (DemoIdentity.jks / DemoTrust.jks) pre-loaded with self-signed, non-production certificates — useful only for quick local testing, never for production.

Custom Identity / Custom Trust

The production approach: pointing WebLogic explicitly at your own identity keystore (with a real, CA-signed certificate) and your own trust keystore (with the appropriate trusted CA certificates), instead of relying on the demo files.

Node Manager SSL

The Node Manager is the WebLogic process responsible for starting/stopping/monitoring managed servers. It has its own, separate SSL configuration to secure its communication channel with the Administration Server.

Admin SSL / Managed Server SSL

The Administration Server (which hosts the WebLogic console) and each Managed Server (which actually runs applications) each have independent SSL settings — identity keystore, trust keystore, and listen ports — that must be configured for each one individually.

Hostname Verification

A setting that controls whether WebLogic checks that the hostname in a certificate actually matches the hostname it connected to. Disabling this (sometimes done for internal testing) removes an important protection against man-in-the-middle attacks and should not be done in production.

SSL Listen Port

The dedicated network port WebLogic listens on for HTTPS/TLS-protected traffic, separate from its plain, unencrypted listen port.

Keystore Providers

WebLogic can be configured to source its identity/trust material from different underlying providers — for example, plain keystore files, or a Hardware Security Module (HSM) — depending on organizational security requirements.

Production vs Development Mode

WebLogic domains run in either Development mode (looser defaults, demo certificates tolerated, easier debugging) or Production mode (stricter security defaults, demo certificates should be replaced, hostname verification enforced). Every real deployment should run in Production mode with custom identity/trust configured.

Module 20 - Troubleshooting

A field guide to the errors you will actually run into, why they happen, and how to resolve them.

PKIX path building failed

Cause: the client cannot build a complete, trusted chain from the server's certificate back to a trusted root — usually because an intermediate certificate is missing.

Solution: install the full chain (server + intermediate) on the server side, or import the missing intermediate/root into the client's truststore.

Unable to find valid certification path

Cause: essentially the same root issue as above — the Java client's truststore doesn't contain a CA it needs to complete the chain.

Solution: import the correct root/intermediate CA certificates into the relevant truststore (e.g. cacerts) using keytool -importcert.

certificate_unknown

Cause: the server rejected a certificate presented to it (often in mutual TLS) because it does not recognize or trust the issuing CA.

Solution: ensure the CA that signed the client certificate is present in the server's trust keystore.

Hostname verification failed

Cause: the hostname you connected to does not match any entry in the certificate's SAN list.

Solution: reissue the certificate with the correct SAN entries for every hostname it needs to serve, or connect using a hostname that matches an existing SAN entry.

Keystore was tampered with, or password was incorrect

Cause: either the keystore password is genuinely wrong, or the keystore file has been corrupted/modified.

Solution: double-check the password being supplied; if it's confirmed correct, restore the keystore from a known-good backup.

Alias does not exist

Cause: a command referenced an alias name that isn't actually present in the keystore (often a typo or case mismatch).

Solution: run keytool -list -v to see the exact alias names actually stored, and correct the command.

No private key

Cause: an operation that requires a private key was pointed at an entry that only contains a public certificate (a "trusted certificate entry"), not a full key pair.

Solution: confirm you're referencing the correct alias that actually holds the private key entry.

Bad Certificate

Cause: the presented certificate is malformed, corrupted, or otherwise fails basic structural validation during the handshake.

Solution: regenerate or re-export the certificate, and verify it independently with openssl x509 -text before deploying it.

Handshake Failure

Cause: a generic failure during negotiation — commonly no overlapping TLS version or cipher suite between client and server.

Solution: check the TLS versions and cipher suites enabled on both sides and ensure they have at least one option in common.

Unsupported protocol

Cause: the client attempted to use a TLS/SSL version that the server has disabled (e.g. a client stuck on TLS 1.0 talking to a server that only allows TLS 1.2+).

Solution: upgrade the client to support a modern TLS version, or explicitly allow the needed version on the server if it is otherwise justified and secure.

No cipher suites in common

Cause: client and server both support TLS, but they don't share any single cipher suite in common.

Solution: expand the enabled cipher suite list on one side (favoring modern, secure suites) so an overlapping option exists.

SunCertPathBuilderException

Cause: the Java-specific exception thrown when the JVM cannot build a valid certificate path — effectively Java's version of the "PKIX path building failed" error.

Solution: same fix as the PKIX error above — import the missing CA certificate into the JVM's truststore (cacerts or a custom truststore).

Module 21 - Best Practices

- Use PKCS12 for new deployments — it's the modern, cross-platform, OpenSSL-compatible standard.
- Use RSA 2048-bit keys or higher (or ECDSA where supported) — 1024-bit is no longer acceptable.
- Use SHA-256 or stronger for signatures — never SHA-1 or MD5.
- Protect keystore passwords — never hard-code them in plain text in source control; use a secrets manager or vault where possible.
- Keep private keys secure — they should never leave the server/system that generated them, and access should be tightly restricted.
- Renew certificates before expiry — track expiration dates proactively rather than reacting to an outage.
- Disable TLS 1.0/1.1 where possible — restrict servers to TLS 1.2 and TLS 1.3 only.
- Verify certificate chains after installation — always confirm with openssl s_client or openssl verify that the full chain resolves correctly.
- Regularly back up keystores and truststores — losing a private key means the certificate must be reissued from scratch.