CHECKLIST

Node.js TLS Hardening Checklist for Secure Production APIs

A vendor-neutral production checklist for hardening TLS in Node.js APIs, covering protocol versions, certificate validation, private key handling, cipher choices, mTLS, and deployment verification.

Node.js TLS Hardening Checklist for Secure Production APIs

Use this checklist to review, validate, and document the TLS posture of a Node.js API before production release.

1) Confirm the TLS termination point

- [ ] Identify where TLS terminates: Node.js process, reverse proxy, ingress controller, API gateway, or load balancer.
- [ ] Document every hop between client and application.
- [ ] Confirm which hop is responsible for certificate presentation and renewal.
- [ ] Verify whether internal service-to-service traffic is encrypted end to end or only at the edge.
- [ ] Record the owner for TLS configuration and certificate rotation.

2) Set a modern protocol baseline

- [ ] Allow only modern TLS versions supported by your runtime and clients.

- [ ] Disable obsolete protocol versions.
- [ ] Verify the runtime behavior in production-like conditions instead of assuming defaults.
- [ ] Confirm no legacy compatibility setting is widening the accepted protocol range.
- [ ] Test a connection attempt using an older protocol to ensure it fails.

3) Validate certificate identity and chain trust

- [ ] Ensure the server certificate matches the hostname clients actually use.
- [ ] Verify the full certificate chain is served correctly, including intermediates.
- [ ] Confirm the issuing CA is trusted by all intended clients.
- [ ] Check that internal DNS names, service names, or SAN entries cover all required endpoints.
- [ ] Confirm certificates are not expired and will remain valid through the release window.
- [ ] Validate that certificate renewal will not change the hostname or trust path unexpectedly.

4) Protect private keys

- [ ] Store private keys in a protected secret store or equivalent secure mechanism.
- [ ] Do not commit private keys to source control.
- [ ] Avoid baking long-lived private keys into immutable images when rotation is expected.
- [ ] Restrict file permissions so only the runtime can read the key.
- [ ] Limit access to deployment logs, backups, and artifacts that may contain key material.
- [ ] Define a rotation process for key replacement and certificate reissue.

5) Review cipher and curve configuration

- [ ] Prefer runtime defaults unless you have a clear security, compliance, or interoperability reason to override them.
- [ ] Avoid legacy cipher lists copied from old examples or blog posts.
- [ ] Confirm that any custom cipher settings still support all required clients.
- [ ] Check whether custom curve settings are necessary; if not, leave them to safe defaults.
- [ ] Test negotiation with the oldest client you explicitly support.
- [ ] Verify no deprecated or unintended suites are allowed by configuration drift.

6) Enforce certificate verification on clients

- [ ] Ensure Node.js clients validate the server certificate chain.
- [ ] Ensure Node.js clients verify the server hostname.
- [ ] Confirm verification is not disabled in code, scripts, or environment-specific overrides.
- [ ] Review third-party SDKs and internal libraries for insecure TLS shortcuts.
- [ ] Reject configurations that use permissive trust settings outside tightly controlled test environments.
- [ ] Test that an untrusted certificate causes the client call to fail.

7) Assess mutual TLS needs

- [ ] Decide whether mTLS is required for this API or only for specific internal paths.
- [ ] Confirm both client and server certificate lifecycle management is operationally ready.
- [ ] Verify each side has the correct trust store and issuing CA.
- [ ] Validate revocation, rotation, and rollout procedures before production use.
- [ ] Keep mTLS separate from application authentication; do not use it as the only control unless that is explicitly intended.

8) Check deployment and infrastructure consistency

- [ ] Confirm TLS settings are consistent across development, staging, and production where intended.
- [ ] Verify load balancers, ingress controllers, and proxies do not reintroduce weak settings.
- [ ] Check for environment variables, config maps, or deployment templates that can silently weaken TLS.
- [ ] Confirm secret mount paths and permissions are correct in containers and hosts.
- [ ] Review health checks and monitoring probes to ensure they use the correct protocol and trust settings.

9) Test failure cases before release

- [ ] Expired certificate: client fails closed.
- [ ] Wrong hostname: client fails closed.
- [ ] Untrusted CA: client fails closed.
- [ ] Old protocol version: connection is rejected.
- [ ] Missing intermediate certificate: connection fails and the issue is observable.
- [ ] Invalid private key or certificate pairing: service startup or TLS handshake fails clearly.

10) Document operational ownership

- [ ] Record who owns certificate issuance and renewal.
- [ ] Record who owns the Node.js runtime and TLS settings.
- [ ] Record how emergencies are handled if a certificate is about to expire.
- [ ] Record how compatibility exceptions are approved and reviewed.

- [ ] Record when the last TLS review was completed.

11) Production go-live signoff

- [ ] TLS terminates where expected and is documented.
- [ ] Only approved protocol versions are accepted.
- [ ] Certificate chain and hostname validation pass in all supported clients.
- [ ] Private keys are stored and accessed securely.
- [ ] Cipher and curve settings are either defaulted or explicitly approved.
- [ ] Any mTLS requirement is implemented and tested.
- [ ] Failure-mode tests passed.
- [ ] Certificate rotation ownership is assigned.

Quick decision rule

If a client cannot connect without disabling verification, accepting an older protocol, or bypassing trust checks, treat that as a compatibility defect to fix?not as a reason to weaken the TLS posture of the API.

Final review questions

- [ ] Can you explain exactly where TLS ends?
- [ ] Can every client verify the server identity successfully?
- [ ] Would the API fail closed if trust is broken?
- [ ] Is certificate rotation operationally ready?
- [ ] Have you tested the most likely failure cases?

Recommended next step

Run this checklist against one production-like environment and one real client integration. Fix the first failing control before expanding the review to the rest of the stack.