HTTPS/TLS: Securing the Modern Internet

A Comprehensive Lecture on Transport Layer Security

Transport Layer Security

Table of Contents

Introduction

Every time you see a **padlock icon** in your browser's address bar, you're witnessing **Transport Layer Security (TLS)** at work.

TLS secures billions of internet connections daily, protecting:

- Online banking
- Private messages
- E-commerce transactions
- API communications

Why TLS Matters

- ✅ **Protects sensitive data** from eavesdropping
- ✅ **Prevents tampering** with transmitted information
- ✅ **Authenticates servers** (and optionally clients)
- ✅ **Essential for user trust** and regulatory compliance
- ✅ **Improves SEO rankings** for websites

What is TLS?

Transport Layer Security (TLS) is a cryptographic protocol designed to provide secure communication over computer networks.

It sits between:

- **Application layer** (like HTTP)
- **Transport layer** (like TCP)

Creating an **encrypted tunnel** for data transmission.

Three Pillars of TLS Security

🔒 Encryption (Confidentiality)

- Hides data from third parties during transmission
- Uses symmetric encryption for performance
- Protects against eavesdropping attacks

✅ Authentication

- Ensures parties are who they claim to be
- Uses digital certificates and public key cryptography
- Prevents impersonation attacks

🛡️ Integrity

- Verifies data hasn't been tampered with
- Uses cryptographic hashing and MACs
- Detects any modification during transmission

Where TLS is Used

- **HTTPS**: Web traffic (the most common use case)
- **Email**: SMTP, IMAP, POP3 connections
- **VoIP**: Secure voice communications
- **VPNs**: Virtual private networks
- **APIs**: RESTful services and microservices
- **IoT**: Device-to-server communication

Historical Context: From SSL to TLS

SSL Era (1995-1999)

- **SSL 1.0**: Never publicly released (security flaws)
- **SSL 2.0** (1995): Developed by Netscape, serious vulnerabilities
- **SSL 3.0** (1996): Major redesign, broken by POODLE attack (2014)

TLS Era (1999-Present)

- **TLS 1.0** (1999): IETF standardized version (SSL 3.1)
- **TLS 1.1** (2006): Fixed weaknesses, CBC attack protection
- **TLS 1.2** (2008): Modern cipher suites, SHA-256 support
- **TLS 1.3** (2018): Major overhaul with improved security and performance

Why "TLS" and Not "SSL"?

The protocol was renamed from **SSL** to **TLS** when the **Internet Engineering Task Force (IETF)** took over standardization from Netscape.

Important: All versions of SSL are now considered insecure and should never be used.

However, the terms are often used interchangeably in common usage, and you'll still see "SSL certificate" referring to what are actually TLS certificates.

Core Security Principles

Cryptographic Building Blocks

1. **Asymmetric (Public Key) Cryptography**
    - Used during handshake phase
    - Examples: RSA, ECDHE, EdDSA
2. **Symmetric Encryption**
    - Used for bulk data encryption
    - Examples: AES-GCM, ChaCha20-Poly1305
3. **Hash Functions**
    - Create fixed-size fingerprints
    - Examples: SHA-256, SHA-384
4. **Key Derivation Functions**
    - Generate keys from shared secrets
    - TLS 1.3 uses HKDF

The Problem: Sending Secrets Over Public Networks

The Challenge

Imagine you want to send a secret message, but you can only communicate through **postcards that anyone can read**.

How do you:

- Share a secret key to encrypt messages?
- Verify you're talking to the right person?
- Ensure no one can read or modify messages?

This is exactly the problem every internet connection faces!

Traditional Encryption Problems

Symmetric Encryption (like AES)

Problem: Both parties need the same secret key
Challenge: How do you share the key without someone stealing it?

It's a **chicken-and-egg problem**!

Why Pre-Sharing Doesn't Work

- Billions of connections daily
- Connecting to servers you've never contacted before
- Impossible to pre-share keys with every website

The Breakthrough: Public Key Cryptography

Revolutionary Ideas (1970s)

- **Asymmetric Encryption**: Two different keys
  - Public key (shared openly)
  - Private key (kept secret)
- **Key Exchange**: Create shared secret without directly transmitting it
- **Digital Signatures**: Prove identity without revealing private keys

The Magic

- Encrypt with public key → Only private key can decrypt
- Sign with private key → Anyone can verify with public key
- Exchange key material publicly → Both parties compute same secret

Real-World Analogy: The Paint Mixing Method

Diffie-Hellman Key Exchange Explained

1. Alice and Bob publicly agree on a common paint color (yellow)
2. Each secretly picks their own color (Alice: red, Bob: cyan)
3. Each mixes their secret color with the common color
4. They publicly exchange these mixed colors
5. Each adds their own secret color to the received mixture
6. **Both end up with the same final color**

Key Insight: The final color was never transmitted!

Diffie-Hellman Key Exchange: The Mathematical Foundation

The Core Mathematical Concept

If we have:

- `A = g^a mod p`
- `B = g^b mod p`

We can calculate `g^(ab) mod p` without revealing the secret exponents `a` or `b`.

This relies on:

- ✅ Modular exponentiation is **easy** to compute
- ❌ Discrete logarithm problem is **extremely hard** to reverse

Step-by-Step: How Diffie-Hellman Works

Step 1: Public Agreement

- `p` (modulus): Large prime number (e.g., 23)
- `g` (base): Primitive root of p (e.g., 5)

Step 2: Secret Selection

- Alice's secret `a` : 6 (never shared)
- Bob's secret `b` : 15 (never shared)

Step 3: Public Exchange

- Alice computes: `A = 5^6 mod 23 = 8`
- Bob computes: `B = 5^15 mod 23 = 19`
- They exchange these values publicly

Step-by-Step: How Diffie-Hellman Works (cont.)

Step 4: Secret Computation

Alice computes:

```
s = 19^6 mod 23 = 2
```

Bob computes:

```
s = 8^15 mod 23 = 2
```

Both got the same shared secret: 2!

Step 5: Why It Works

- Alice: `(g^b)^a = g^(b×a) mod p`
- Bob: `(g^a)^b = g^(a×b) mod p`
- Since multiplication is commutative: `g^(b×a) = g^(a×b)`

Security: The Discrete Logarithm Problem

Why Can't Eve Find the Secret?

Eve knows:

- `p = 23`
- `g = 5`
- `A = 8` (which is `5^a mod 23`)
- `B = 19` (which is `5^b mod 23`)

To find the shared secret, Eve needs to:

- Calculate `a` from `5^a mod 23 = 8`, OR
- Calculate `b` from `5^b mod 23 = 19`

With real numbers: Modern DH uses at least **2048-bit numbers** (617 decimal digits), making brute force computationally infeasible.

Real-World Implementation Sizes

Modern Diffie-Hellman implementations use much larger numbers:

- **2048-bit**: 617 decimal digits (minimum recommended)
- **3072-bit**: 925 decimal digits
- **4096-bit**: 1234 decimal digits

Example 2048-bit prime has over **600 digits**!

Elliptic Curve Diffie-Hellman (ECDHE)

Benefits Over Traditional DH

- ✅ **Shorter keys** for equivalent security
  - 256-bit ECDHE ≈ 3072-bit DH
- ✅ **Faster computation**
- ✅ **Less bandwidth**

How It Differs

- Uses points on elliptic curves instead of modular exponentiation
- Based on **Elliptic Curve Discrete Logarithm Problem (ECDLP)**
- Operations are point addition/multiplication on the curve

Common Curves

- **X25519** (Curve25519) - recommended
- P-256 (secp256r1)
- P-384 (secp384r1)

Ephemeral Diffie-Hellman (DHE/ECDHE)

What is "Ephemeral"?

Ephemeral means "short-lived":

- New key pair generated for each connection
- Keys discarded after session ends
- Provides **Perfect Forward Secrecy (PFS)**

Perfect Forward Secrecy

Even if server's private key is compromised later:

- ✅ Past recorded traffic remains secure
- ✅ Each session has unique encryption keys

TLS 1.3 requires ephemeral key exchange for all connections.

Vulnerability: Man-in-the-Middle

The Attack

```
Alice  <---> Mallory (attacker) <---> Bob
```

- Alice thinks she's doing DH with Bob
- Bob thinks he's doing DH with Alice
- Actually, both are doing DH with Mallory!

Solution: Authentication

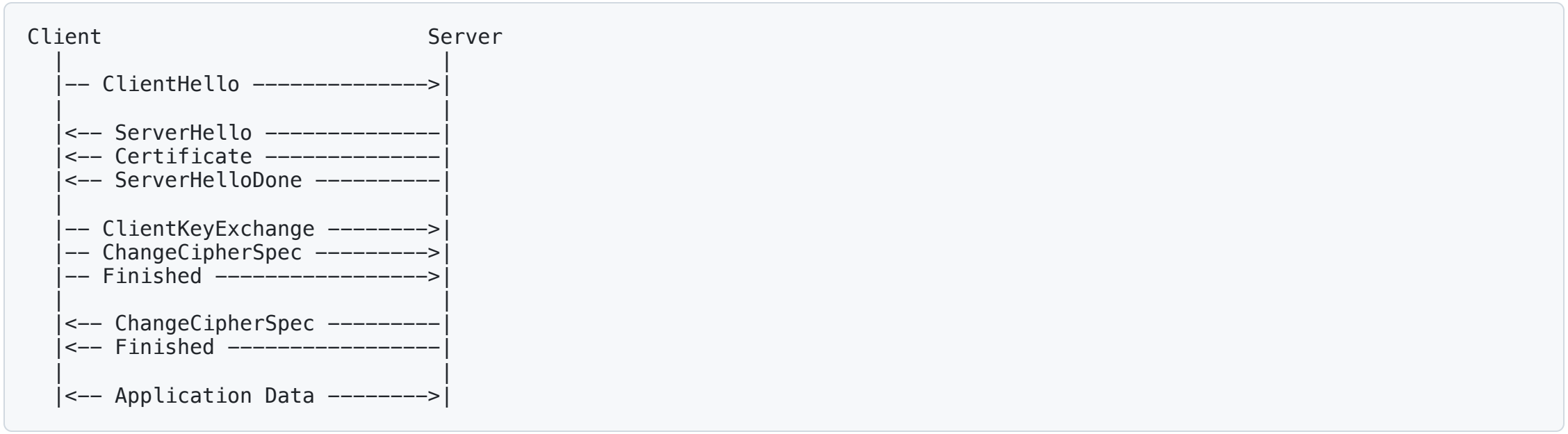
- Server certificates prove server identity
- Signed with Certificate Authority's private key
- Client verifies signature with CA's public key
- **Combines DH key exchange with authentication**

This is exactly what TLS does!

How TLS Works: The Handshake Protocol

The TLS handshake is a carefully choreographed exchange that establishes a secure connection.

TLS 1.2 Handshake (Traditional)



Round Trips: TLS 1.2 requires **2 round trips (2-RTT)** before application data can flow.

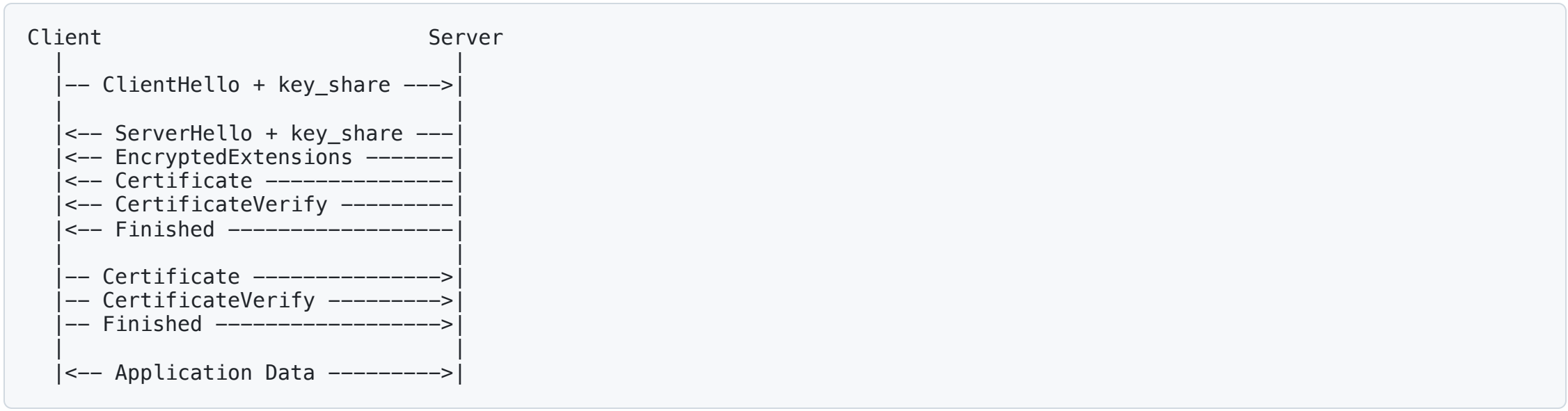
TLS 1.2 Handshake: Step-by-Step

1. **ClientHello**: Client initiates connection
    - Supported TLS versions
    - List of cipher suites
    - Random number
    - Server name (SNI)
2. **ServerHello**: Server responds
    - Selected TLS version
    - Chosen cipher suite
    - Server random number
    - Session ID
3. **Certificate**: Server sends TLS certificate
    - Server's public key
    - Server identity information
    - CA signature

TLS 1.2 Handshake: Step-by-Step (cont.)

4. **ClientKeyExchange**: Client generates and sends
    - Pre-master secret (encrypted with server's public key) OR
    - Ephemeral Diffie-Hellman parameters
5. **Key Generation**: Both sides independently compute
    - Master secret from pre-master secret and random values
    - Session keys for encryption and MAC
6. **ChangeCipherSpec & Finished**: Both parties
    - Switch to encrypted communication
    - Send verification messages encrypted with new keys
7. **Application Data**: Secure communication begins

TLS 1.3 Handshake (Modern)



Key Improvement: **1-RTT handshake** - Application data can be sent after just one round trip!

TLS 1.3: Key Improvements

1. **1-RTT Handshake**
    - Application data after one round trip
    - **0-RTT mode** for resumed connections (with security trade-offs)
2. **Always Forward Secret**
    - Mandatory use of ephemeral key exchange
    - No static RSA
3. **Simplified Cipher Suites**
    - Removed outdated and insecure options
4. **Encrypted Handshake**
    - Most handshake messages are now encrypted

Certificates and Public Key Infrastructure

What is a TLS Certificate?

A TLS certificate (X.509 certificate) is a digital document that:

- ✅ Binds a public key to a specific domain/organization
- ✅ Is cryptographically signed by a trusted Certificate Authority
- ✅ Contains metadata about the certificate holder
- ✅ Has an expiration date

Certificate Validation Levels

1. Domain Validated (DV)

- **Verification**: Only proves control of domain
- **Speed**: Issued within minutes
- **Cost**: Often free (Let's Encrypt)
- **Use Case**: Blogs, portfolios, non-commercial sites

2. Organization Validated (OV)

- **Verification**: Validates organization legitimacy
- **Speed**: Days to weeks
- **Cost**: Moderate ($50-$200/year)
- **Use Case**: Business websites, e-commerce

3. Extended Validation (EV)

- **Verification**: Rigorous identity verification
- **Speed**: Weeks
- **Cost**: Higher ($100-$500+/year)
- **Use Case**: Banks, high-security sites

Certificate Authorities (CAs)

Top Certificate Authorities

- **Let's Encrypt** (free, automated)
- DigiCert
- IdenTrust
- Sectigo
- GlobalSign

The Trust Chain

```
Root CA Certificate (pre-installed in OS/browser)
    ↓ signs
Intermediate CA Certificate
    ↓ signs
End-Entity Certificate (your website)
```

Why Intermediates?

- Root keys kept offline for security
- Intermediate compromise doesn't require replacing root

Certificate Best Practices

- ✅ **Key Size**: Minimum 2048-bit RSA keys (4096-bit preferred)
- ✅ **Hash Algorithm**: Use SHA-256 or better (never MD5 or SHA-1)
- ✅ **Validity Period**: Maximum 398 days (13 months) as of 2020
- ✅ **Subject Alternative Names**: Include all relevant domains
- ✅ **Certificate Transparency**: Publicly log all certificates
- ✅ **OCSP Stapling**: Allow servers to provide revocation status

TLS 1.3: The Modern Standard

TLS 1.3 (RFC 8446) represents a fundamental redesign, published in August 2018 after nearly five years of development and 28 drafts.

Major Improvements

1. **Security Enhancements**
2. **Performance Improvements**
3. **Simplified Cipher Suite Selection**
4. **Improved Privacy**

TLS 1.3: Security Enhancements

Removed Insecure Features

- ❌ RSA key exchange (no forward secrecy)
- ❌ Static Diffie-Hellman
- ❌ CBC mode ciphers (vulnerable to padding oracle attacks)
- ❌ RC4, 3DES, MD5, SHA-1
- ❌ Compression (vulnerable to CRIME attack)
- ❌ Renegotiation

Mandatory Security

- ✅ Perfect Forward Secrecy (PFS) required for all connections
- ✅ Authenticated encryption (AEAD) only
- ✅ Improved key derivation using HKDF
- ✅ Better protection against downgrade attacks

TLS 1.3: Performance Improvements

Faster Connections

- **1-RTT handshake** (down from 2-RTT)
- **0-RTT mode** for resumed connections
- Reduced latency improves user experience

Benchmark Impact

- ~30% faster handshake completion
- Significant improvement on high-latency connections
- Better for mobile devices

TLS 1.3: Simplified Cipher Suites

TLS 1.2 Cipher Suite Example

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
```

Specifies: key exchange + authentication + encryption + MAC

TLS 1.3 Cipher Suite Example

```
TLS_AES_128_GCM_SHA256
```

Only specifies: encryption + hash (key exchange and auth negotiated separately)

Supported TLS 1.3 Cipher Suites

- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_AES_128_GCM_SHA256
- TLS_AES_128_CCM_SHA256
- TLS_AES_128_CCM_8_SHA256

TLS 1.3: Improved Privacy

Encrypted Handshake

- Most handshake messages now encrypted
- Hides certificate information from observers
- Better protects metadata

Server Name Indication (SNI) Encryption

- Addressed in **Encrypted Client Hello (ECH)** extension
- Prevents passive observers from seeing which site you're visiting

0-RTT: The Double-Edged Sword

TLS 1.3's **0-RTT mode** allows clients to send application data in the very first message when resuming a connection.

Benefits

- ✅ Zero added latency for resumed connections
- ✅ Excellent for frequently accessed sites

Security Trade-offs

- ⚠️ Vulnerable to replay attacks
- ⚠️ Should only be used for idempotent operations
- ⚠️ Many implementations disable it by default

Recommended Use

- ✅ **Safe**: GET requests for public data
- ❌ **Unsafe**: POST requests, authentication, state-changing operations

HTTPS: TLS in Action

HTTPS = HTTP + TLS

It's the secure version of HTTP, operating by default on:

- **Port 443** (vs. port 80 for HTTP)

How Browsers Show Security

✅ Secure Connection

- Padlock icon displayed
- "https://" in address bar
- Can click padlock to view certificate

⚠️ Problems

- "Not Secure" warning for HTTP sites
- Broken padlock or warning for certificate issues
- Browser blocks access for serious security problems

HTTP Strict Transport Security (HSTS)

HSTS forces browsers to always use HTTPS for a domain.

HSTS Header Example

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

Benefits

- ✅ Prevents protocol downgrade attacks
- ✅ Protects against SSL stripping
- ✅ Users can't bypass certificate warnings
- ✅ Can be preloaded into browsers

Implementing HTTPS: Step-by-Step

1. **Obtain a Certificate**
    - Choose validation level (DV/OV/EV)
    - Select Certificate Authority
    - Generate Certificate Signing Request (CSR)
    - Complete domain validation
    - Receive and install certificate
2. **Configure Web Server**
    - Install certificate and private key
    - Configure TLS settings
    - Set up certificate chain
    - Enable modern cipher suites
    - Configure HSTS

Implementing HTTPS: Step-by-Step (cont.)

3. **Update Application**
    - Change all internal links to HTTPS
    - Update external resources (CSS, JS, images)
    - Fix mixed content warnings
    - Redirect HTTP to HTTPS (301 permanent redirect)
4. **Test and Monitor**
    - Validate certificate installation
    - Test browser compatibility
    - Scan for security issues
    - Monitor certificate expiration
    - Check performance metrics

Best Practices: Protocol Versions

Recommended Configuration

- ✅ **Enable**: TLS 1.3, TLS 1.2
- ❌ **Disable**: TLS 1.1, TLS 1.0, SSL 3.0, SSL 2.0

Rationale

- TLS 1.2 required for older client compatibility
- TLS 1.3 for modern clients and optimal security
- Older protocols have known vulnerabilities

Best Practices: Cipher Suite Selection

Modern Cipher Suite Priority (TLS 1.2)

```
ssl_ciphers 'ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305';
```

Principles

- ✅ Prioritize AEAD ciphers (GCM, ChaCha20-Poly1305)
- ✅ Require forward secrecy (ECDHE)
- ❌ Avoid CBC mode ciphers when possible
- ❌ No RC4, 3DES, or export ciphers

Performance Optimization

1. **Session Resumption**
   - Enable session caching
   - Use session tickets carefully (consider privacy implications)
2. **OCSP Stapling**
   - Reduces client-side revocation checks
   - Improves performance and privacy
3. **HTTP/2 and HTTP/3**
   - HTTP/2 requires TLS (in practice)
   - HTTP/3 uses QUIC (includes TLS 1.3)
   - Multiplexing reduces connection overhead
4. **Certificate Chain Optimization**
   - Include all intermediate certificates
   - Minimize chain length

Common Vulnerabilities: Historical TLS Attacks

Understanding past vulnerabilities helps us secure the present:

1. **BEAST** (2011): TLS 1.0 CBC ciphers
2. **CRIME** (2012): TLS compression
3. **POODLE** (2014): SSL 3.0
4. **Heartbleed** (2014): OpenSSL buffer over-read
5. **FREAK** (2015): Export-grade cryptography
6. **Logjam** (2015): Weak DH parameters
7. **DROWN** (2016): SSL 2.0 cross-protocol attack
8. **Sweet32** (2016): 64-bit block ciphers (3DES)
9. **ROBOT** (2017): RSA PKCS#1 v1.5
10. **Lucky Thirteen** (2013): CBC mode timing attack

Modern Security Considerations

1. Certificate Transparency

- All certificates logged publicly
- Detects unauthorized certificate issuance
- Monitor CT logs for your domains

2. CAA Records

```
example.com. CAA 0 issue "letsencrypt.org"
example.com. CAA 0 issuewild ";"
example.com. CAA 0 iodef "mailto:security@example.com"
```

- Specifies which CAs can issue certificates
- Prevents unauthorized issuance

Modern Security Considerations (cont.)

3. Certificate Pinning

- Hard-codes expected certificates or public keys
- Effective for mobile apps
- HPKP deprecated for web, but still useful for apps

4. Monitoring and Alerting

- Certificate expiration monitoring
- SSL/TLS configuration scanning
- Security vulnerability alerts
- Performance monitoring

Testing and Validation: Online Tools

1. SSL Labs Server Test

- **URL**: https://www.ssllabs.com/ssltest/
- Comprehensive security analysis
- Grades A+ through F
- Detailed protocol and cipher testing
- Certificate chain validation
- Vulnerability scanning

2. testssl.sh

- Command-line tool
- Checks protocols, ciphers, vulnerabilities
- Highly detailed output
- Can test any TLS service

Browser Developer Tools

- **Chrome DevTools**: Security tab → View certificate, protocol, cipher suite
- **Firefox Developer Tools**: Network tab → Select request → Security tab

Ideal Configuration Checklist

- ✅ TLS 1.3 and/or TLS 1.2 only
- ✅ Strong cipher suites (AEAD only)
- ✅ Certificate from trusted CA
- ✅ Complete certificate chain
- ✅ Certificate not expired
- ✅ HSTS enabled
- ✅ OCSP stapling working
- ✅ No known vulnerabilities
- ✅ A or A+ grade on SSL Labs
- ✅ Forward secrecy enabled

Resources: Official Standards

IETF RFCs

- **RFC 8446** - TLS 1.3 - Official TLS 1.3 specification
- **RFC 5246** - TLS 1.2 - TLS 1.2 specification
- **RFC 6101** - SSL 3.0 - Historical reference (do not use)

NIST Guidelines

- **SP 800-52 Rev. 2** - Guidelines for TLS Implementations

Resources: Configuration Guides

Mozilla SSL Configuration Generator

- **URL**: https://ssl-config.mozilla.org/
- Generate server configs for Apache, Nginx, more
- Choose between Modern, Intermediate, Old profiles

Security Organizations

- **OWASP TLS Cheat Sheet** - Comprehensive best practices
- **Cloudflare Learning Center** - Educational resources
- **MDN Web Docs** - TLS - Developer-focused documentation

Resources: Testing and Monitoring Tools

SSL Labs

- https://www.ssllabs.com/ssltest/ - Server test
- https://www.ssllabs.com/ssltest/viewMyClient.html - Client test

Command-Line Tools

- **testssl.sh** - Comprehensive TLS testing
- **sslscan** - Fast SSL/TLS scanner
- **nmap NSE scripts** - Network scanning

Certificate Monitoring

- **Certificate Transparency Logs** - Search CT logs
- **Cert Spotter** - Monitor for unauthorized certificates

Thank You!

Questions?

HTTPS/TLS: Securing the Modern Internet

Transport Layer Security