Cryptographic Protocols in Web Development

MAT364 - Cryptography Course

Instructor: Adil Akhmetov

University: SDU

Week 10

Press Space for next page →

Week 10 Focus

Motivation

- Web apps move secrets (tokens, cookies, credentials)
- Attacks (MITM, downgrade, CSRF) target weak crypto plumbing
- Goal: deploy TLS, sessions, APIs with provable guarantees

Learning Outcomes

1. Explain TLS 1.3 handshake flow and certificate validation
2. Implement HTTPS-only services with mutual trust anchors
3. Secure API tokens (JWT/OAuth2) and browser storage

Agenda

- TLS 1.3 and HTTPS deployment
- Secure cookies, sessions, CSRF defence
- OAuth2/OIDC + JWT best practices
- API gateways, mTLS, WebSockets security
- Lab: Harden an Express API + Python client

TLS 1.3 Deep Dive

TLS 1.3 Handshake Flow

Timeline

1. **ClientHello**: cipher suites, key shares (X25519/P-256)
2. **ServerHello**: picks params, sends certificate + CertificateVerify
3. **Finished** messages authenticated with HKDF-derived keys
4. **Application Data** encrypted via AEAD (AES-GCM/ChaCha20-Poly1305)

Key Material

- HKDF-Extract(ECDHE, salt) → Handshake Secret
- HKDF-Expand → Client/Server handshake keys
- After Finished: derive Application traffic keys and resumption Master Secret

Python mTLS Snippet (`ssl` + `httpx`)

```
import httpx, ssl

def create_tls_context():
    ctx = ssl.create_default_context(ssl.Purpose.SERVER
    ctx.minimum_version = ssl.TLSVersion.TLSv1_3
    ctx.set_ciphers("TLS_AES_256_GCM_SHA384:TLS_CHACHA2
    ctx.load_verify_locations(cafile="ca.pem")
    ctx.load_cert_chain(certfile="client.crt", keyfile=
    return ctx

with httpx.Client(verify=create_tls_context()) as clien
    resp = client.get("https://api.internal.sdu")
    resp.raise_for_status()
    print(resp.json())
```

Takeaways

- Disable TLS 1.0/1.1, prefer TLS 1.3
- Pin CA bundle for internal services
- Monitor cert expiry + OCSP stapling

HTTPS Deployment Checklist

Layer	Requirement	Tooling
Certificates	Automated issuance + rotation	ACME/Let’s Encrypt, Smallstep CA
Cipher Suites	Forward secrecy + AEAD	TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256
HSTS	`Strict-Transport-Security: max-age=63072000; includeSubDomains; preload`	Nginx, Cloudflare
OCSP	Stapled responses, `must-staple`	certbot with `--staple-ocsp`, AWS ACM
Logging	JA3 fingerprints, failed handshakes	Envoy, Istio, OpenTelemetry

Reminder: Everything behind auth must still enforce HTTPS and reject plain HTTP at load balancers and origins.

Secure Sessions & Cookies

Cookies, Sessions, CSRF

Cookie Flags

- `Secure` : HTTPS-only transport
- `HttpOnly` : blocks JS access, mitigates XSS theft
- `SameSite=Lax/Strict` : CSRF defence
- `Domain` + `Path` : limit scope

Session Stores

- Rotate session IDs after login
- Store minimal PII; encrypt values at rest
- Use short TTL (≤24h) + sliding expiration

Express Middleware Example

```
import express from "express";
import session from "express-session";
import helmet from "helmet";
import csrf from "csurf";

const app = express();
app.use(helmet({ contentSecurityPolicy: false }));
app.set("trust proxy", 1);

app.use(session({
  name: "mat364.sid",
  secret: process.env.SESSION_SECRET!,
  resave: false,
  saveUninitialized: false,
  cookie: {
```

CSRF Tokens

- Synchronizer token pattern (`csrf()` middleware)
- Double-submit cookie for SPAs
- Combine with SameSite=Lax for external redirects

OAuth2, OIDC & JWT

OAuth2 Grant Types

Flow	Use Case	Security Notes
Authorization Code + PKCE	Mobile/SPA clients	PKCE protects code interception, use short-lived auth codes
Client Credentials	Service-to-service APIs	Keep client secret in vault/HSM, scope tokens narrowly
Device Code	TVs, CLI	Rate-limit polling, expire device codes quickly
Refresh Tokens	Long-lived sessions	Store encrypted, bind to client/device, rotate on use

OIDC Add-ons: `id_token` (user identity) with nonce, `userinfo` endpoint, discovery document (`.well-known/openid-configuration`).

JWT Implementation Patterns

Signing Keys

- Prefer EdDSA (`Ed25519`) or ES256
- Use `kid` header + JWKS endpoint for rotation
- Keep private keys offline/HSM; publish only JWKS JSON

Validation Checklist

1. Verify signature with expected alg
2. Check `iss` , `aud` , `sub`
3. Enforce `exp` , `nbf` , `iat` windows
4. Check `jti` against replay cache if needed

Node.js Verification Helper

```
import { jwtVerify, createRemoteJWKSet } from "jose";

const JWKS = createRemoteJWKSet(new URL("https://auth.r

export async function verifyAccessToken(token: string)
  const { payload } = await jwtVerify(token, JWKS, {
    issuer: "https://auth.mat364.sdu",
    audience: "lecture-api",
    algorithms: ["RS256", "ES256", "EdDSA"]
  });
  if (payload.scope?.includes("admin")) {
    enforceMFA(payload);
  }
  return payload;
}
```

Storage Guidance

- Access token → in-memory (React state, Redux store)
- Refresh token → HttpOnly cookie with SameSite=Strict
- Never store tokens in `localStorage`

API Gateways & Service Mesh

Mutual TLS & Zero Trust

Components

- **Identity Provider:** issues SPIFFE IDs
- **Proxy/Mesh:** Envoy/Istio handles cert rotation
- **Policy Engine:** OPA/Styra define authZ

mTLS Workflow

1. Sidecar fetches short-lived cert from CA
2. Mutual handshake occurs per request
3. Envoy injects verified peer identity header
4. Application enforces RBAC on identity claims

Envoy Filter Snippet (YAML)

```
transport_socket:
  name: envoy.transport_sockets.tls
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.trans
    common_tls_context:
      tls_params:
        tls_minimum_protocol_version: TLSv1_3
      tls_certificates:
        certificate_chain: { filename: "/etc/envoy/cert
        private_key: { filename: "/etc/envoy/certs/serv
      validation_context:
        trusted_ca: { filename: "/etc/envoy/certs/mesh-
        match_typed_subject_alt_names:
          - san_type: DNS
            matcher: { exact: "service-a.mesh.sdu" }
```

Observability

- Export TLS metrics (`ssl.handshake` , `ssl.connection_error`)
- Collect distributed traces with identity labels

WebSockets & Real-Time Channels

Securing WebSockets

Requirements

- Use `wss://` only; same TLS profile as HTTPS
- Authenticate upgrade request (token/cookie)
- Re-validate token periodically (ping/pong)
- Rate-limit connection attempts per IP/user

Threats

- Stolen bearer tokens reused indefinitely
- Downgrade to ws:// via mixed content
- Message injection without per-message auth

Node Server with Token Binding

```
import { WebSocketServer } from "ws";
import { verifyAccessToken } from "./jwt";

const wss = new WebSocketServer({ noServer: true });

wss.on("connection", (socket, ctx) => {
  const { user, exp } = ctx.tokenPayload;
  const ttl = (exp * 1000) - Date.now();
  const refreshInterval = Math.min(ttl / 2, 5 * 60 * 10

  const interval = setInterval(async () => {
    try {
      ctx.tokenPayload = await verifyAccessToken(ctx.to
    } catch {
      socket.close(4003, "Token expired");
```

Client Tips

- Store token per tab, refresh via secure iframe/postMessage
- Fail closed (auto disconnect on verification error)

Lab: Harden a Web API

🎯 Student Lab Assignment

Scenario

You inherit an Express + PostgreSQL API that currently serves HTTP and stores JWTs in `localStorage`. The goal is to make it production-ready.

Tasks

1. Enable TLS 1.3 via Nginx reverse proxy with automatic certificates.
2. Move refresh tokens into HttpOnly cookies and rotate them on each use.
3. Enforce OAuth2 Authorization Code + PKCE for the frontend SPA.
4. Implement CSRF protection for state-changing routes.
5. Add security headers (`helmet`) and CSP that allows only self + CDN fonts.

Deliverables

- Updated server configuration + `docker-compose` snippet
- Postman/HTTPie collection showing successful auth flow
- Short write-up describing threat mitigations

✅ Solution Outline

Infrastructure

- Use Caddy or Nginx with `certbot --deploy-hook systemctl reload`
- Redirect port 80 → 443, enable HSTS preload
- Add mutual TLS between gateway and internal API

Auth Flow

1. SPA hits `/oauth/authorize` → receives code + PKCE verifier
2. Backend exchanges code for tokens, sets refresh cookie (`SameSite=Strict`)
3. Access token returned in JSON (in-memory use)
4. Refresh endpoint rotates cookie, invalidates old token in DB

Sample Nginx Snippet

```
server {
  listen 443 ssl http2;
  server_name api.mat364.sdu;

  ssl_protocols TLSv1.3;
  ssl_ciphers TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1
  add_header Strict-Transport-Security "max-age=630720

  location / {
    proxy_pass http://api_internal;
    proxy_set_header X-Forwarded-Proto https;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwa
    proxy_set_header X-Client-Cert $ssl_client_cert;
  }
}
```

Testing Matrix

- TLS scan (sslyze/Qualys) must score A+
- Automated integration tests for token rotation + CSRF failures

Best Practices & Pitfalls

Operational Checklist

- **Secrets Management:** store API keys, client secrets, session keys in Vault/AWS Secrets Manager; rotate quarterly
- **Monitoring:** alert on TLS certificate expiry, failed handshakes, 4xx auth spikes
- **Logging:** log `sub`, `jti`, `cid` (client ID) with privacy-safe hashing
- **Defense in Depth:** combine WAF rules + rate limiting + anomaly detection
- **Incident Response:** rehearse key compromise playbooks (revoke certs, rotate JWKS, invalidate refresh tokens)

Anti-patterns to avoid: mixed-content pages, storing secrets in source control, sharing tokens across browser tabs via \`localStorage\`, ignoring certificate pinning warnings.

Summary

- TLS 1.3 provides modern primitives; enforce it end-to-end
- Cookies and sessions must leverage Secure/HttpOnly/SameSite + rotation
- OAuth2 + OIDC flows require algorithm whitelisting and short-lived tokens
- Mutual TLS + service mesh enables zero-trust internal networks
- WebSockets, APIs, and gateways share the same crypto hygiene expectations

Next Week: Cryptography in mobile applications (secure storage, platform APIs).

Assignment: Ship the lab deliverables + upload OpenSSL scan report.

Questions?

Thanks for exploring web crypto protocols! 🌐🔐