

Day 2: OWASP Top 10 Threats and Mitigation Strategies

Comprehensive Training with Practical Examples and Hands-on Exercises

Security Training Team

Web Application Security Department

November 6, 2025

Agenda Hari 2 (Praktik + Teori)

- Tujuan: Memahami risiko OWASP Top 10 melalui praktik langsung
- Teori ringkas tiap risiko: akar masalah, skenario serangan, dampak
- Lab terarah: langkah eksploitasi aman dan mitigasi terukur
- Review hasil lab: indikator keberhasilan dan cara verifikasi
- Mini-CTF dan rubric penilaian

Prasyarat & Setup Lingkungan

Perangkat yang dibutuhkan

- Docker Desktop
(Windows/Mac/Linux)
- PowerShell atau Terminal Bash
- OWASP ZAP / Burp (opsional)
- Postman atau curl

Garis besar arsitektur

- Aplikasi target: DVWA, Juice Shop, WebGoat
- Tools DAST: OWASP ZAP baseline scan
- Proxy debugging: Burp/ZAP

Variabel lingkungan (contoh)

- DB_PASSWORD, REDIS_PASSWORD
- SESSION_SECRET
- MONGODB_URI / DATABASE_URL

Catatan keamanan

- Jalankan hanya di lingkungan lab terisolasi
- Jangan menggunakan data produksi

Docker Compose: Lab Mandiri

Jalankan seluruh target dan tools secara lokal

```
1 version: '3.8'  
2 services:  
3   dvwa:  
4     image: vulnerables/web-dvwa  
5     ports:  
6       - "8080:80"  
7  
8   juiceshop:  
9     image: bkimminich/juice-shop  
10    ports:  
11      - "3000:3000"  
12  
13  webgoat:  
14    image: webgoat/webgoat-8.2  
15    ports:  
16      - "8081:8080"
```

Metodologi Pengujian

Pendekatan

- SAST: analisis kode statis
- DAST: uji black-box aplikasi berjalan
- IAST: sensor di dalam aplikasi saat runtime
- Threat modeling: STRIDE, data flow

Definisi Keberhasilan

- Dapat mereproduksi kerentanan secara aman
- Menerapkan mitigasi dan memverifikasi perbaikan
- Menulis catatan uji dan rekomendasi

A01: Broken Access Control - Teori

- Akar masalah: pemeriksaan otorisasi tidak konsisten atau hilang
- Pola umum: IDOR, bypass otorisasi, force browsing, mass assignment
- Dampak: kebocoran data, modifikasi data pengguna lain
- Strategi: verifikasi otorisasi server-side di tiap aksi dan objek

A01: Lab - IDOR dan Cek Otorisasi

Target: DVWA (modul File atau View Profile)

- ① Login sebagai user biasa, catat *request* untuk melihat profil:
`/vulnerabilities/view_user.php?id=2`
- ② Ubah parameter id menjadi milik user lain (mis. 1) via Burp Repeater
- ③ Observasi apakah data user lain terlihat (indikator IDOR)
- ④ **Mitigasi:** tambahkan middleware cek kepemilikan sumber daya

Contoh middleware (Express.js)

```
1 // Pastikan resource dimiliki oleh user yang login
2 function authorizeOwnership(getOwnerIdFromResource) {
3   return async (req, res, next) => {
4     try {
5       const resourceId = req.params.id;
6       const ownerId = await getOwnerIdFromResource(resourceId);
7       if (!req.user || String(req.user.id) !== String(ownerId)) {
8         return res.status(403).json({ error: 'Forbidden' });
9     }
10    }
11  }
12 };
```

A01: Snippet Go + JS (ESM)

Go: Middleware kepemilikan (net/http)

```
1 type OwnerFunc func(id string) (
2     string, error)
3
4 func AuthorizeOwnership(getOwner
5     OwnerFunc, next http.Handler)
6     http.Handler {
7         return http.HandlerFunc(func(w
8             http.ResponseWriter, r *http.
9             Request) {
10             userID := r.Header.Get("X-User
11                 -ID") // contoh: hasil auth
12                 sebelumnya
13                 id := r.URL.Query().Get("id")
14                 ownerID, err := getOwner(id)
15                 if err != nil || userID == ""
16             }
```

JS (ESM): Hindari kepercayaan sisi-klien

```
1 // modules/validate.js
2 export function isValidId(v){
3     return /^[0-9]+/.test(v); }
4
5 // app.js (ESM)
6 import {isValidId} from './
7     modules/validate.js';
8 const id = new URLSearchParams(
9     location.search).get('id');
10 if (!isValidId(id)) alert('Invalid
11     ', );
```

A04: Lab - Threat Modeling & Secure Design

Tujuan: Mengidentifikasi abuse case dan kontrol desain yang hilang.

- ① Pilih satu fitur (mis. upload file atau reset password)
- ② Gambar data flow sederhana: sumber data, proses, penyimpanan, trust boundary
- ③ Identifikasi ancaman (STRIDE): spoofing, tampering, repudiation, info disclosure, DoS, elevation of privilege
- ④ Tentukan kontrol: validasi, autentikasi, otorisasi, rate limit, logging, enkripsi
- ⑤ Verifikasi kontrol pada implementasi (code/config review singkat)

Artefak: DFD ringkas + daftar kontrol yang dipilih + rencana uji.

A02: Snippet Go - TLS & HSTS

Go server dengan TLS modern + HSTS

```
1 srv := &http.Server{ Addr: ":443" }
2 http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
3     w.Header().Set("Strict-Transport-Security", "max-age=31536000;
4         includeSubDomains")
5     w.Header().Set("X-Content-Type-Options", "nosniff")
6     w.Header().Set("X-Frame-Options", "DENY")
7     w.Header().Set("Content-Security-Policy", "default-src 'self'")
8     fmt.Fprintln(w, "ok")
9 })
10 // Pastikan sertifikat disediakan; gunakan reverse proxy jika perlu
log.Fatal(srv.ListenAndServeTLS("server.crt", "server.key"))
```

A06: Lab - Dependency & SBOM Scanning

Langkah:

- ① Pindai dependencies proyek contoh (Node/Python/Java)
- ② Hasilkan SBOM dan scan untuk CVE
- ③ Terapkan update minor/patch, verifikasi kompatibilitas

Perintah contoh

```
1 # Node.js
2 npm ci
3 npm audit --audit-level=high || true
4 osv-scanner --lock package-lock.json || true
5
6 # Python
7 pip install -r requirements.txt
8 pip-audit || true
9
10 # Java (Maven)
11 mvn -DskipTests org.owasp:dependency-check-maven:check || true
```

A08: Lab - Supply Chain Integrity

Langkah:

- ① Verifikasi integritas artefak container dengan Sigstore Cosign
- ② Validasi signature webhook (HMAC) di aplikasi
- ③ Kunci dependency ke lockfile dan aktifkan verifikasi di CI

Cosign verify (contoh)

```
1 COSIGN_EXPERIMENTAL=1 cosign verify ghcr.io/org/app:latest \
2   --certificate-oidc-issuer https://token.actions.githubusercontent.com \
3   --certificate-identity "https://github.com/org/repo/.github/workflows/release.yml@refs/heads/main"
```

Go modules di CI

```
1 govulncheck ./...
2 go mod verify
```

Verifikasi HMAC (Node.js)

A09: Lab - Structured Logging & Alerts

Langkah:

- ① Tambahkan structured logging untuk event keamanan (auth, akses ditolak, input error)
- ② Kirim log ke file/stdout; uji rotasi/retensi
- ③ Buat aturan alert sederhana untuk 5x gagal login/menit

Contoh (Node.js - winston)

```
1 const winston = require('winston');
2 const log = winston.createLogger({
3   level: 'info',
4   format: winston.format.json(),
5   transports: [ new winston.transports.Console() ]
6 });
7 function logAuth(username, success) {
8   log.info({ event: 'auth', username, success, ts: Date.now() });
9 }
```

A02: Cryptographic Failures - Teori

- Prinsip: enkripsi in transit (TLS) dan at rest, manajemen kunci
- Anti-pola: HTTP tanpa TLS, algoritma lemah, kunci disimpan di repo
- Dampak: pencurian kredensial, manipulasi data
- Strategi: TLS modern, HSTS, rotasi kunci, secret manager

A02: Lab - HTTPS, HSTS, dan Header

- ① Akses login DVWA melalui HTTP, tangkap kredensial via ZAP (lab-only)
- ② Aktifkan TLS pada reverse proxy dan pasang HSTS
- ③ Verifikasi dengan curl -I https://localhost dan periksa header

Contoh Nginx (ringkas)

```
1 server {  
2     listen 443 ssl;  
3     ssl_protocols TLSv1.2 TLSv1.3;  
4     add_header Strict-Transport-Security "max-age=31536000;  
5         includeSubDomains" always;  
6     add_header X-Content-Type-Options nosniff;  
7     add_header X-Frame-Options DENY;  
8     add_header Content-Security-Policy "default-src 'self'";  
9     location / { proxy_pass http://dvwa:80; }  
}
```

A03: Snippet Go - Prepared Statement

database/sql + driver MySQL

```
1 import (
2     "database/sql"
3     _ "github.com/go-sql-driver/mysql"
4 )
5
6 db, _ := sql.Open("mysql", os.Getenv("MYSQL_DSN"))
7 id, _ := strconv.Atoi(r.URL.Query().Get("id"))
8 row := db.QueryRowContext(r.Context(), "SELECT id, name FROM users
9 WHERE id = ?", id)
10 var user struct{ ID int; Name string }
11 if err := row.Scan(&user.ID, &user.Name); err != nil { /* handle */
    }
12 json.NewEncoder(w).Encode(user)
```

A03: Injection - Teori

- Pola: SQLi, NoSQLi, command injection, LDAP injection
- Akar masalah: konkatenasi input ke query/command tanpa sanitasi
- Strategi: prepared statements, validasi input, least privilege DB

A03: Lab - SQLi pada DVWA dan Mitigasi

- ① DVWA: menu SQL Injection, payload awal: ' OR 1=1 --
- ② Amati hasil enumerasi data
- ③ **Mitigasi:** gunakan prepared statements

Contoh (Node.js - mysql2)

```
1 const mysql = require('mysql2/promise');
2 const pool = mysql.createPool({ uri: process.env.DATABASE_URL });
3 app.get('/user', async (req, res) => {
4   const id = parseInt(req.query.id, 10);
5   const [rows] = await pool.execute('SELECT * FROM users WHERE id = ?',
6     [id]);
7   res.json(rows[0] || {});
});
```

A05: Snippet Go + HTML (ESM/CSP)

Go: Middleware header keamanan

```
1 func SecurityHeaders(next http.Handler)
2     http.Handler {
3         return http.HandlerFunc(func(w http.
4             ResponseWriter, r *http.Request) {
5                 w.Header().Set("Content-Security-
6                     Policy", "default-src 'self'")
7                 w.Header().Set("X-Content-Type-
8                     Options", "nosniff")
9                 w.Header().Set("X-Frame-Options", "DENY")
10                w.Header().Set("Referrer-Policy", "no-
11                    referrer")
12                next.ServeHTTP(w, r)
13            })
14        }
```

HTML: ESM + CSP

```
1 <meta http-equiv="Content-
2     Security-Policy"
3         content="default-src
4             'self'; script-src 'self
5                 '; object-src 'none'">
6 <script type="module" src="
7     /app.js"></script>
```

A05: Security Misconfiguration - Teori

- Umum: default credential, directory listing, pesan error verbose
- Header keamanan tidak aktif, CORS terlalu permissif
- Strategi: hardening, automation, baseline konfigurasi

A05: Lab - Header Keamanan dan Hardening

- ① Cek header: curl -I http://localhost:8080
- ② Tambahkan Helmet pada Express atau header di Nginx
- ③ Verifikasi ulang header terpasang

Express + Helmet

```
1 const helmet = require('helmet');
2 app.use(helmet({
3   contentSecurityPolicy: { useDefaults: true },
4   hsts: { maxAge: 31536000, includeSubDomains: true }
5 }));
```

A07: Snippet Go - Rate Limit & Cookie

Rate limit sederhana per IP

```
1 var hits = make(map[string]int)
2 var windowStart = time.Now()
3 func RateLimit(next http.Handler) http.
4 Handler {
5     return http.HandlerFunc(func(w http.
6 ResponseWriter, r *http.Request) {
7         if time.Since(windowStart) > time.
8 Minute { hits = map[string]int{}; }
9         windowStart = time.Now() }
10        ip, _, _ := net.SplitHostPort(r.
11 RemoteAddr)
12        hits[ip]++
13        if hits[ip] > 100 { http.Error(w, "
14 too many requests", 429); return }
15        next.ServeHTTP(w, r)
16    })
17 }
```

Cookie aman

```
1 http.SetCookie(w, &http.
2   Cookie{
3       Name: "session", Value:
4           token,
5       HttpOnly: true, Secure:
6           true,
7       SameSite: http.
8           SameSiteStrictMode,
9       })
10 }
```

A07: Auth Failures - Teori

- Masalah: password lemah, bruteforce, session fixation, token tidak aman
- Strategi: MFA, rate limiting, bcrypt/argon2, cookie HttpOnly + Secure

A07: Lab - Rate Limit, Password Policy, Session

- ① Uji brute force ringan ke endpoint login
- ② Tambah rate limit dan lockout setelah N percobaan gagal
- ③ Audit flag cookie: HttpOnly, Secure, SameSite

Contoh Rate Limit (Express)

```
1 const rateLimit = require('express-rate-limit');
2 const limiter = rateLimit({ windowMs: 15*60*1000, max: 100 });
3 app.use('/login', limiter);
```

A10: Snippet Go - Validasi URL (SSRF)

Allowlist skema + blok IP privat

```
1 func isPrivate(ip net.IP) bool {
2     private := []string{"10.0.0.0/8", "172.16.0.0/12", "192.168.0.0/16"
3         , "127.0.0.0/8"}
4     for _, cidr := range private {
5         _, n, _ := net.ParseCIDR(cidr); if n.Contains(ip) { return true
6     }
7 }
8
9 func validateURL(raw string) bool {
10    u, err := url.Parse(raw); if err != nil { return false }
11    if u.Scheme != "http" && u.Scheme != "https" { return false }
12    ips, err := net.LookupIP(u.Hostname()); if err != nil { return
13        false }
14    for _, ip := range ips { if isPrivate(ip) { return false } }
```

Frontend ESM: Validasi & Output Encoding

Validasi input (ESM)

```
1 // modules/validators.js
2 export const isEmail = s => /^[^@\s]+@[^\s]+\.[^@\s]+$/ .test(s);
3 export const isSafeLen = (s, n
4   =100) => s.length <= n;

1 // app.js
2 import { isEmail, isSafeLen } from
3   './modules/validators.js';
4 const email = document.
5   querySelector('#email').value;
6 if(!isEmail(email) || !isSafeLen(
7   email, 120)) alert('Invalid
8   email');
```

Output encoding aman

```
1 const mount = document.
2   getElementById('greeting');
3 const name = new URLSearchParams(
4   location.search).get('name') ||
5   '';
6 // Jangan gunakan innerHTML untuk
7   data yang tidak dipercaya
8 mount.textContent = 'Hello, ${name
9   }'; // aman:textContent
```

A10: SSRF - Teori

- Pola: server melakukan fetch ke URL input pengguna
- Risiko: akses jaringan internal, metadata cloud, pemindaian internal
- Strategi: allowlist skema/host, blok alamat private, SSRF proxy

A10: Lab - Validasi URL dan Blok Jaringan Internal

- ① WebGoat: pelajaran SSRF, coba akses endpoint internal
- ② Terapkan validator URL dengan allowlist dan blok RFC1918
- ③ Verifikasi akses ke host internal ditolak

Validator (Node.js)

```
1 const { isIP } = require('net');
2 function isPrivateIp(host) {
3   return /^10\.|^192\.168\.|^172\.(1[6-9]|2[0-9]|3[0-1])\./.test(
4     host);
5 }
6 function validateUrl(input) {
7   try {
8     const u = new URL(input);
9     if (!['http:', 'https:'].includes(u.protocol)) return false;
10    if (isIP(u.hostname) && isPrivateIp(u.hostname)) return false;
11    return true;
12  } catch { return false; }
```

Run Labs (Go Service)

- Jalankan snippet Go (A05/A07/A10) sebagai service lokal
- Perintah cepat: go mod tidy, go run .
- Uji header: curl -I http://localhost:PORT
- Uji rate limit: loop curl -j 100 req/menit, pastikan 429
- Uji SSRF: endpoint fetch URL hanya menerima skema HTTP/HTTPS publik
- Gunakan env: SESSION_SECRET, DATABASE_URL bila diperlukan

Proxy ZAP/Burp untuk ESM

- Set proxy browser: 127.0.0.1:8080 (ZAP) atau 127.0.0.1:8081 (Burp)
- Import CA ZAP/Burp ke browser agar HTTPS dapat diinspeksi
- Buka aplikasi target, jalankan skenario dari slide (login, cari, upload)
- Gunakan Repeater untuk memodifikasi parameter (IDOR/SQLi)
- Jalankan Baseline/Active Scan (ZAP) pada `http://localhost:...`
- Verifikasi mitigasi: header aktif, query terparametrisasi, blok SSRF

Mini-CTF: Tugas Akhir

- Selesaikan 5 tantangan di Juice Shop (kategori Injection, Broken Access)
- Tulis laporan singkat: bukti, dampak, mitigasi
- Sertakan bukti verifikasi perbaikan (header, rate limit, prepared statements)

Rubrik Penilaian

- Reproduksi kerentanan (30%)
- Implementasi mitigasi (40%)
- Dokumentasi dan verifikasi (20%)
- Kolaborasi/komunikasi (10%)

Training Objectives

Learning Outcomes:

- ✓ Understand OWASP Top 10 security risks
- ✓ Identify vulnerabilities in web applications
- ✓ Apply effective mitigation strategies
- ✓ Implement security best practices

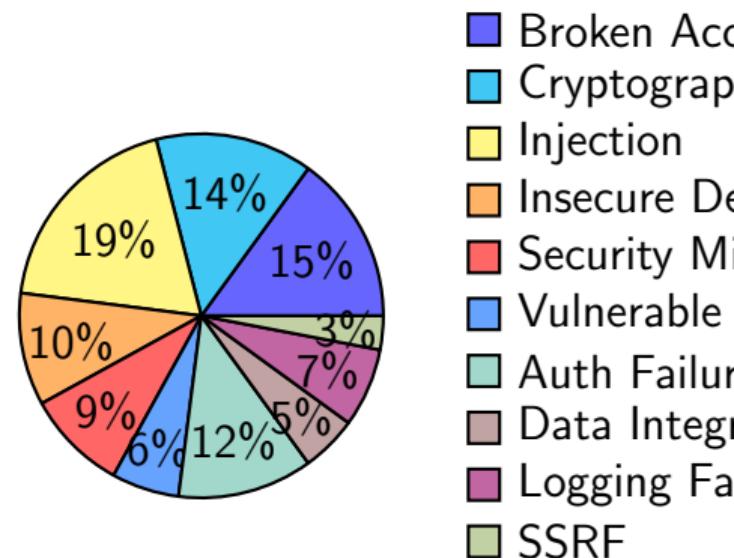
Practical Skills:

-  Vulnerability assessment techniques
-  Security testing methodologies
-  Code security implementation
-  Hands-on vulnerability labs

OWASP Top 10 2021 Overview

The OWASP Top 10 represents the most critical security risks to web applications:

Rank	Risk Category
A01:2021	Broken Access Control
A02:2021	Cryptographic Failures
A03:2021	Injection
A04:2021	Insecure Design
A05:2021	Security Misconfiguration
A06:2021	Vulnerable and Outdated Components
A07:2021	Identification and Authentication Failures
A08:2021	Software and Data Integrity Failures
A09:2021	Security Logging and Monitoring Failures
A10:2021	Server-Side Request Forgery (SSRF)



A01:2021 - Broken Access Control

What is Broken Access Control?

- Users can access resources or perform actions beyond their intended permissions
- Most common security vulnerability in web applications
- Affects 94% of applications tested
- Can lead to data breaches and privilege escalation

Common Attack Vectors:

- ⚠ Insecure direct object references (IDOR)
- ⚠ Missing access control checks

Vulnerable Code Example:

```
1 // Vulnerable: No access control
  check
2 app.get('/api/users/:id', (req,
  res) => {
  3   const userId = req.params.id;
  4   const user = db.getUserById(
    userId);
  5   res.json(user); // Any user
  6   can access any user data
  7 });
  8 // Vulnerable: Parameter tampering
  9 app.post('/api/admin/delete-user',
  10   (req, res) => {
  11     const userId = req.body.userId
  12   };
  13 }
```

A01:2021 - Broken Access Control: Mitigation

Secure Implementation:

```
// Secure: Role-based access
control
app.get('/api/users/:id',
  requireAuth,
  requireRole(['admin', 'user']),
(req, res) => {
  const userId = req.params.id;

  // Check if user can access
  this resource
  if (req.user.role !== 'admin'
    &&
      req.user.id !== userId) {
    return res.status(403).json
  (
    error: 'Access denied'
```

Best Practices:

① Implement Access Control

- ✓ Role-based access control (RBAC)
- ✓ Attribute-based access control (ABAC)

A02:2021 - Cryptographic Failures

Vulnerable Code Example:

```
1 // Vulnerable: Weak hashing
  algorithm
2 const bcrypt = require('bcrypt');
3 const password = 'user123';
4
5 // Weak: Using MD5 (deprecated)
6 const weakHash = require('crypto')
7   .createHash('md5')
8   .update(password)
9   .digest('hex');
10
11 // Vulnerable: Hardcoded
  encryption key
12 const crypto = require('crypto');
13 const algorithm = 'aes-256-cbc';
14 const key = '
```

What are Cryptographic Failures?

- Sensitive data exposure due to weak cryptography
- Affects 44% of applications tested
- Can lead to identity theft, financial loss, and data breaches

Common Issues:

A02:2021 - Cryptographic Failures: Mitigation

Secure Implementation:

```
1 // Secure: Strong password hashing
2 const bcrypt = require('bcrypt');
3 const password = 'user123';
4
5 // Strong: Using bcrypt with
6 // appropriate cost factor
7 const saltRounds = 12;
8 const strongHash = bcrypt.hashSync
9 (password, saltRounds);
10
11 // Secure: Proper key management
12 const crypto = require('crypto');
13 const algorithm = 'aes-256-gcm';
14 const keyLength = 32; // 256 bits
15 const ivLength = 16; // 128 bits
16
```

Vulnerable Code Example:

```
1 // Vulnerable: SQL Injection
2 app.get('/api/user/:id', (req, res)
3   ) => {
4     const userId = req.params.id;
5     const query = 'SELECT * FROM
6       users WHERE id = ${userId}';
7     db.query(query, (err, results)
8       => {
9       res.json(results);
10      });
11    }
12
13 // Vulnerable: Command Injection
14 app.post('/api/download', (req,
15   res) => {
16   const filename = req.body.
```

What is Injection?

- Attackers can inject malicious code or data into your application

A03:2021 - Injection: Mitigation

Secure Implementation:

```
1 // Secure: Parameterized queries (SQL)
2 const mysql = require('mysql2/promise');
3
4 app.get('/api/user/:id', async (
5   req, res) => {
6   const userId = req.params.id;
7
8   try {
9     const [rows] = await db.
10    execute(
11      'SELECT * FROM users
12      WHERE id = ?',
13      [userId] //
14      Parameterized query
```

A04:2021 - Insecure Design

What is Insecure Design?

- Security is not considered during the design phase
- Results in fundamental security flaws that are difficult to fix
- Affects 23% of applications tested

Common Design Flaws:

- ⚠ Missing threat modeling
- ⚠ Insecure data flow design
- ⚠ Weak authentication flows
- ⚠ Lack of security by design

Design Principles:

① Zero Trust Architecture

- Never trust, always verify
- Micro-segmentation
- Continuous authentication

② Defense in Depth

- Multiple security layers
- Redundant controls
- Fail-safe mechanisms

③ Security by Design

- Threat modeling
- Risk assessment
- Secure coding standards

④ Privacy by Design

A04:2021 - Insecure Design: Mitigation

Secure Design Patterns:

```
1 // Secure: Authentication flow
2   design
3 const express = require('express')
4 ;
5 const session = require('express-
6   session');
7 const crypto = require('crypto');
8
9 const app = express();
10
11 // Secure session configuration
12 app.use(session({
13   secret: process.env.
14     SESSION_SECRET,
15   resave: false,
16   saveUninitialized: false,
```

A05:2021 - Security Misconfiguration

Vulnerable Configuration:

```
1 # Vulnerable: Nginx
   misconfiguration
2 server {
3     listen 80;
4     server_name example.com;
5
6     # Missing security headers
7     # Missing SSL/TLS
8     configuration
9     # Default directory listing
10    enabled
11    autoindex on;
12
13    # Verbose error messages
14    error_page 404 /404.html;
15    error_page 500 /500.html;
```

What is Security Misconfiguration?

- Security settings are not properly implemented or maintained
- Affects 19% of applications tested
- Often due to lack of security

A05:2021 - Security Misconfiguration: Mitigation

Secure Configuration:

```
1 # Secure: Nginx configuration
2 server {
3     listen 443 ssl http2;
4     server_name example.com;
5
6     # SSL/TLS configuration
7     ssl_certificate /etc/ssl/certs
8     /cert.pem;
9     ssl_certificate_key /etc/ssl/
10    private/key.pem;
11    ssl_protocols TLSv1.2 TLSv1.3;
12    ssl_ciphers HIGH:!aNULL:!MD5:!13
13    RC4;
14    ssl_prefer_server_ciphers on;
15
16    # Security headers
```

A06:2021 - Vulnerable and Outdated Components

Vulnerable Dependencies:

```
1 // package.json with vulnerable  
2 // dependencies  
3 {  
4   "name": "vulnerable-app",  
5   "version": "1.0.0",  
6   "dependencies": {  
7     "express": "4.16.0", //  
8     // vulnerable to multiple CVEs  
9     "lodash": "4.17.11", //  
// prototype pollution  
// vulnerability  
10    "moment": "2.22.2", //  
11    // prototype pollution  
12    // vulnerability  
13    "ws": "6.2.1", // vulnerable  
14    // to WebSocket attacks
```

What are Vulnerable Components?

- Using third-party libraries with known vulnerabilities
- Affects 17% of applications tested
- Can lead to complete system compromise

A06:2021 - Vulnerable Components: Mitigation

Secure Dependency Management:

```
// package.json with secure  
dependencies  
{  
  "name": "secure-app",  
  "version": "1.0.0",  
  "dependencies": {  
    "express": "^4.18.2", //  
      latest secure version  
    "lodash": "^4.17.21", //  
      patched version  
    "moment": "^2.29.4", //  
      patched version  
    "ws": "^8.13.0", // latest  
      secure version  
    "mysql2": "^3.6.0" // secure  
      MySQL driver
```

Vulnerable Authentication:

```
1 // Vulnerable: Weak password
  hashing
2 const crypto = require('crypto');
3
4 function hashPassword(password) {
5   // Weak: Using MD5 (deprecated)
6   return crypto.createHash('md5')
7     .update(password).digest('hex');
8
9 // Vulnerable: Insecure session
10 management
11 const express = require('express')
12 ;
13 const session = require('express-
```

A07:2021 - Authentication Failures: Mitigation

Secure Authentication:

```
1 // Secure: Strong password hashing
2 const bcrypt = require('bcrypt');
3
4 async function hashPassword(
5   password) {
6   // Strong: Using bcrypt with
7   // appropriate cost factor
8   const saltRounds = 12;
9   return await bcrypt.hash(
10     password, saltRounds);
11 }
12
13 // Secure: Secure session
14 management
15 const express = require('express')
16 ;
```

A08:2021 - Software and Data Integrity Failures

Vulnerable Code:

```
1 // Vulnerable: Insecure
   deserialization
2 const { deserialize } = require(
  'vulnerable-parser');
3
4 function processData(data) {
5   // Dangerous: Direct
6   // deserialization without
7   // validation
8   const obj = deserialize(data);
9   return obj;
10
11 // Vulnerable: Missing integrity
12   checks
13 const fs = require('fs');
```

A08:2021 - Integrity Failures: Mitigation

Secure Implementation:

```
1 // Secure: Safe deserialization
2 const { safeDeserialize } =
3   require('secure-parser');
4
5 function processData(data) {
6   try {
7     // Validate input before
8     // deserialization
9     if (!isValidSerializedData(
10       data)) {
11       throw new Error('Invalid
12       serialized data');
13     }
14
15     // Use safe deserialization
16     const obj = safeDeserialize(
17       data);
18   }
19 }
```

A09:2021 - Security Logging and Monitoring Failures

Vulnerable Logging:

```
1 // Vulnerable: Insufficient
  logging
2 const express = require('express')
  ;
3 const app = express();
4
5 app.post('/api/login', (req, res)
  => {
6   const { username, password } =
  req.body;
7
8   // No logging of login attempts
9   const user = db.getUser(username
  , password);
10
11  if (user) {
```

A09:2021 - Logging Failures: Mitigation

Secure Logging:

```
1 // Secure: Comprehensive logging
2 const winston = require('winston')
3   ;
4 const { combine, timestamp, printf
5       } = winston.format;
6
7 // Custom log format
8 const logFormat = printf(({ level,
9     message, timestamp }) => {
10   return `${timestamp} [${level}]:` +
11     `${message}`;
12 });
13
14 // Create logger instance
15 const logger = winston.
16   createLogger({
```

A10:2021 - Server-Side Request Forgery (SSRF)

Vulnerable SSRF Code:

```
1 // Vulnerable: User-controlled URL  
2 // requests  
3  
4 const axios = require('axios');  
5  
6 app.get('/api/fetch-url', (req,  
7 res) => {  
8     const url = req.query.url;  
9  
10    // Dangerous: Direct request to  
11    // user-provided URL  
12    axios.get(url)  
13        .then(response => {  
14            res.json(response.data);  
15        })  
16        .catch(error => {  
17            res.status(500).json({ error  
18                ...  
19            })  
20        })  
21    })  
22});
```

What is SSRF?

- Attackers can force the server to make

A10:2021 - SSRF: Mitigation

Secure SSRF Prevention:

```
1 // Secure: URL validation and
2   restrictions
3
4 const axios = require('axios');
5 const url = require('url');
6 const { URL } = require('url');
7
8 // Allowed domains list
9 const ALLOWED_DOMAINS = [
10   'api.example.com',
11   'trusted-service.com',
12   'internal-service.local'
13 ];
14
15 // Validate URL against allowed
16   domains
17 function isUrlAllowed(targetUrl) {
18   const parsedUrl = new URL(targetUrl);
19   return ALLOWED_DOMAINS.includes(parsedUrl.hostname);
20 }
```

Hands-on Lab: Vulnerability Assessment and Mitigation

Lab Objectives:

- ① Identify vulnerabilities in web applications
- ② Assess risk levels of identified threats
- ③ Develop mitigation strategies
- ④ Create security testing reports

Lab Environment Setup:

- ① Install Docker and vulnerable applications
- ② Set up security scanning tools
- ③ Configure logging and monitoring
- ④ Prepare test cases and payloads

Lab Tasks:

- ① **Vulnerability Scanning**
 - Run automated vulnerability scans
 - Identify OWASP Top 10 vulnerabilities
 - Document findings with evidence
 - Prioritize risks based on impact
- ② **Manual Testing**
 - Perform penetration testing
 - Test for injection attacks
 - Verify access control issues
 - Test authentication bypasses
- ③ **Mitigation Implementation**
 - Apply security patches
 - Implement input validation

Practical Example: Secure Web Application

Docker Compose for Security Testing:

```
1 version: '3.8'  
2  
3 services:  
4   # Target application  
5   vulnerable-app:  
6     image: vulnerable-web-app:  
7       latest  
8       ports:  
9         - "8080:80"  
10      environment:  
11        - DB_HOST=postgres  
12        - REDIS_HOST=redis  
13      depends_on:  
14        - postgres  
15        - redis
```

Secure Application Features:

① Authentication and Authorization

- ✓ Multi-factor authentication

Security Assessment Checklist

OWASP Top 10 Assessment Checklist:

① Broken Access Control

- ✓ Role-based access implemented
- ✓ Authorization checks enforced
- ✓ Insecure direct object references prevented
- ✓ Access control tested

② Cryptographic Failures

- ✓ Strong encryption algorithms used
- ✓ Secure key management
- ✓ Transport security implemented
- ✓ Sensitive data protected

③ Injection

- ✓ Parameterized queries used

Security Configuration Checklist:

① Security Misconfiguration

- ✓ Default credentials changed
- ✓ Security headers configured
- ✓ Error messages secured
- ✓ Services properly configured

② Vulnerable Components

- ✓ Dependencies scanned
- ✓ Known vulnerabilities patched
- ✓ Outdated components updated
- ✓ Secure alternatives used

③ Authentication Failures

- ✓ Strong password policies
- ✓ Multi-factor authentication

Summary and Key Takeaways

OWASP Top 10 Fundamentals:

-  Understanding security risks
-  Identifying vulnerabilities
-  Applying mitigation strategies
-  Implementing security controls

Security Best Practices:

-  Defense in depth
-  Principle of least privilege
-  Security by design
-  Continuous security testing

Practical Implementation:

-  Secure coding practices
-  Security tool integration
-  Regular security assessments
-  Documentation and policies

Next Steps:

-  Apply concepts to real projects
-  Conduct security audits
-  Implement security controls
-  Continuous improvement

Questions?