# Sécurité des Applications Web - Jour 4

m2iformation.fr

# Sécurité des Applications Web

## Jour 4 : Sécurisation du Développement

### Bonnes Pratiques et Protections

# Objectifs du Jour 4

- Appliquer les recommandations OWASP pour chaque vulnérabilité
- Implémenter la validation des entrées
- Configurer les headers de sécurité HTTP
- Sécuriser les sessions et cookies
- Protéger les API et Web Services

# Module 1

## Ressources OWASP

# Les Guides OWASP

| Guide | Contenu |
|---|---|
| **Testing Guide** | Méthodologie de test de sécurité |
| **Code Review Guide** | Revue de code sécurité |
| **ASVS** | Application Security Verification Standard |
| **Cheat Sheet Series** | Fiches pratiques par sujet |

# Outils d'apprentissage

- **WebGoat** : Application volontairement vulnérable
- **Juice Shop** : Application moderne avec vulnérabilités OWASP

# ASVS - Niveaux de Vérification

| Niveau | Description | Cible |
|---|---|---|
| **Level 1** | Opportunistic | Toutes applications |
| **Level 2** | Standard | Applications avec données sensibles |
| **Level 3** | Advanced | Applications critiques |

Le niveau 2 est recommandé pour la plupart des applications manipulant des données personnelles ou financières.

# Module 2

# Protection contre Broken Access Control

# Principes de Protection

## Deny by default

```
# MAUVAIS : autoriser par défaut
def check_access(user, resource):
    if resource in user.denied_resources:
        return False
    return True  # Autorisé si non explicitement refusé

# BON : refuser par défaut
def check_access(user, resource):
    if resource in user.allowed_resources:
        return True
    return False  # Refusé si non explicitement autorisé
```

Toujours partir du principe que l'accès est interdit sauf autorisation explicite.

# Vérification Coté Serveur

```python
from functools import wraps
from flask import session, abort

def require_permission(permission):
    """
    Décorateur vérifiant les permissions avant exécution.
    A appliquer sur chaque endpoint nécessitant un controle.
    """
    def decorator(func):
        @wraps(func)  # Préserve les métadonnées de la fonction
        def wrapper(*args, **kwargs):
            # Vérifier l'authentification
            if 'user_id' not in session:
                abort(401)  # Non authentifié

            # Vérifier l'autorisation
            user = get_user(session['user_id'])
            if not user.has_permission(permission):
                abort(403)  # Non autorisé

            return func(*args, **kwargs)
        return wrapper
    return decorator
```

# Application du Décorateur

```python
@app.route('/admin/users')
@require_permission('admin.users.view')
def list_users():
    """
    Cette route est protégée par le décorateur.
    Seuls les utilisateurs avec la permission
    'admin.users.view' peuvent y accéder.
    """
    users = User.query.all()
    return render_template('admin/users.html', users=users)

@app.route('/api/documents/<int:doc_id>')
@require_permission('documents.read')
def get_document(doc_id):
    # Vérification supplémentaire : l'utilisateur
    # a-t-il accès à CE document spécifique ?
    doc = Document.query.get_or_404(doc_id)
    if doc.owner_id != session['user_id']:
        abort(403)
    return jsonify(doc.to_dict())
```

# Module 3

## Protection contre les Injections

# Requetes Préparées - Tous Langages

## Python avec SQLAlchemy

```python
from sqlalchemy import text

# VULNERABLE : concaténation
query = f"SELECT * FROM users WHERE id = {user_id}"

# SECURISE : paramètres liés
# Le :user_id est un placeholder remplacé de manière sure
query = text("SELECT * FROM users WHERE id = :user_id")
result = db.execute(query, {"user_id": user_id})

# Encore mieux : utiliser l'ORM
# L'ORM génère automatiquement des requetes sures
user = User.query.filter_by(id=user_id).first()
```

# Requetes Préparées - Java

```java
// VULNERABLE : concaténation de chaines
String query = "SELECT * FROM users WHERE id = " + userId;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);

// SECURISE : PreparedStatement
// Le ? est un placeholder pour le paramètre
String query = "SELECT * FROM users WHERE id = ?";
PreparedStatement pstmt = conn.prepareStatement(query);

// setInt définit le premier paramètre (index 1) comme entier
// Le driver échappe automatiquement la valeur
pstmt.setInt(1, userId);
ResultSet rs = pstmt.executeQuery();
```

# Requetes Préparées - Node.js

```
// VULNERABLE : template string
const query = `SELECT * FROM users WHERE id = ${userId}`;
connection.query(query);

// SECURISE : paramètres positionnels
// Le ? est remplacé par la valeur du tableau
// mysql2 échappe automatiquement les valeurs
const query = 'SELECT * FROM users WHERE id = ?';
const [rows] = await connection.execute(query, [userId]);

// Avec un ORM (Sequelize)
// L'ORM gère les paramètres de manière sure
const user = await User.findOne({
    where: { id: userId }
});
```

# Protection XSS - Echappement

```
// Fonction d'échappement HTML
function escapeHtml(unsafe) {
    // Remplace les caractères spéciaux par leurs entités HTML
    // Cela empêche l'interprétation comme balises
    return unsafe
        .replace(/&/g, "&amp;")   // & en premier (sinon double encodage)
        .replace(/</g, "&lt;")    // < devient &lt;
        .replace(/>/g, "&gt;")    // > devient &gt;
        .replace(/"/g, "&quot;")  // " devient &quot;
        .replace(/'/g, "&#039;"); // ' devient &#039;
}

// Utilisation
const userInput = '<script>alert("XSS")</script>';
const safe = escapeHtml(userInput);
// Résultat : &lt;script&gt;alert(&quot;XSS&quot;)&lt;/script&gt;
// Affiché tel quel, non exécuté
```

# Content Security Policy (CSP)

## Configuration complète

```
Content-Security-Policy:
    default-src 'self';
    script-src 'self' https://cdn.example.com;
    style-src 'self' 'unsafe-inline';
    img-src 'self' data: https:;
    font-src 'self' https://fonts.googleapis.com;
    connect-src 'self' https://api.example.com;
    frame-ancestors 'none';
    base-uri 'self';
    form-action 'self';
```

Chaque directive controle une catégorie de ressources. 'self' autorise uniquement le meme domaine.

# CSP - Directives Expliquées

| Directive | Controle |
|---|---|
| `default-src` | Politique par défaut pour tout |
| `script-src` | Sources JavaScript autorisées |
| `style-src` | Sources CSS autorisées |
| `img-src` | Sources d'images autorisées |
| `connect-src` | Destinations fetch/XHR autorisées |
| `frame-ancestors` | Qui peut intégrer la page (iframe) |
| `form-action` | Destinations des formulaires |

`'unsafe-inline'` autorise le code inline (déconseillé pour script-src).

# Module 4

## Validation des Entrées

# Principes de Validation

## Validation en plusieurs couches

```
[Entrée utilisateur]
        |
        v
[Validation coté client]  <-- UX, pas de sécurité
        |
        v
[Validation coté serveur] <-- Sécurité obligatoire
        |
        v
[Sanitization]            <-- Nettoyage des données
        |
        v
[Stockage/Traitement]
```

La validation client est contournable. La validation serveur est obligatoire.

# Types de Validation

## Validation de format

```python
import re
from email_validator import validate_email

def validate_user_input(data):
    errors = []

    # Email - utiliser une bibliothèque spécialisée
    try:
        validate_email(data['email'])
    except Exception as e:
        errors.append(f"Email invalide: {e}")

    # Téléphone - regex pour format francais
    # ^0[1-9] : commence par 0 suivi d'un chiffre 1-9
    # [0-9]{8}$ : suivi de 8 chiffres
    if not re.match(r'^0[1-9][0-9]{8}$', data['phone']):
        errors.append("Format téléphone invalide")

    return errors
```

# Validation de Type et Plage

```python
def validate_product_data(data):
    errors = []

    # Validation de type
    # isinstance vérifie le type de la variable
    if not isinstance(data.get('quantity'), int):
        errors.append("La quantité doit etre un entier")

    # Validation de plage
    # La quantité doit etre positive et raisonnable
    quantity = data.get('quantity', 0)
    if quantity < 1 or quantity > 1000:
        errors.append("Quantité hors limites (1-1000)")

    # Validation de longueur
    name = data.get('name', '')
    if len(name) < 2 or len(name) > 100:
        errors.append("Nom: 2-100 caractères requis")

    return errors
```

# Sanitization

```python
import bleach
import html

def sanitize_user_content(content):
    """
    Nettoie le contenu utilisateur pour un affichage sur.
    Bleach permet de conserver certaines balises HTML sures.
    """
    # Liste blanche de balises autorisées
    allowed_tags = ['p', 'br', 'strong', 'em', 'ul', 'li']

    # Liste blanche d'attributs autorisés par balise
    allowed_attrs = {'*': ['class']}  # class autorisé partout

    # Nettoyage avec bleach
    # Supprime toutes les balises non autorisées
    # Echappe leur contenu au lieu de les supprimer
    clean = bleach.clean(
        content,
        tags=allowed_tags,
        attributes=allowed_attrs,
        strip=True  # Supprime les balises non autorisées
    )

    return clean
```

# Module 5

## Sécurité des Sessions

# Configuration des Cookies de Session

```python
from flask import Flask
from datetime import timedelta

app = Flask(__name__)

# Clé secrète pour signer les cookies
# Doit etre unique et gardée secrète
app.secret_key = 'clé-très-longue-et-aléatoire-générée-de-manière-sure'

app.config.update(
    # Cookie transmis uniquement en HTTPS
    SESSION_COOKIE_SECURE=True,

    # Cookie inaccessible au JavaScript
    SESSION_COOKIE_HTTPONLY=True,

    # Cookie non envoyé avec les requetes cross-site
    SESSION_COOKIE_SAMESITE='Strict',

    # Durée de vie de la session (30 minutes)
    PERMANENT_SESSION_LIFETIME=timedelta(minutes=30)
)
```

# Protection CSRF

## Génération du token

```python
import secrets
from flask import session

def generate_csrf_token():
    """
    Génère un token CSRF unique pour la session.
    Le token est stocké en session et comparé à chaque POST.
    """
    if 'csrf_token' not in session:
        # secrets.token_hex génère un token aléatoire
        # cryptographiquement sur (32 bytes = 64 caractères hex)
        session['csrf_token'] = secrets.token_hex(32)
    return session['csrf_token']

def validate_csrf_token(token):
    """Valide le token CSRF fourni avec la requete."""
    # Comparaison en temps constant pour éviter les timing attacks
    # hmac.compare_digest compare sans révéler d'information
    import hmac
    return hmac.compare_digest(
        token or '',
        session.get('csrf_token', '')
    )
```

# Intégration CSRF dans les Formulaires

```
<!-- Template HTML avec token CSRF -->
<form method="POST" action="/submit">
    <!-- Champ caché contenant le token CSRF -->
    <!-- csrf_token() est une fonction injectée dans le template -->
    <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">

    <label>Nom:</label>
    <input type="text" name="name">

    <button type="submit">Envoyer</button>
</form>

<!--
Le serveur vérifie que le token soumis correspond
à celui stocké en session. Un attaquant ne peut pas
connaitre ce token car il est unique par session.
-->
```

# Validation CSRF Coté Serveur

```python
from functools import wraps
from flask import request, abort

def csrf_protect(func):
    """
    Décorateur vérifiant le token CSRF sur les requetes POST.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Vérifier uniquement les méthodes qui modifient des données
        if request.method in ['POST', 'PUT', 'DELETE']:
            # Récupérer le token du formulaire ou du header
            token = request.form.get('csrf_token') or \
                    request.headers.get('X-CSRF-Token')

            if not validate_csrf_token(token):
                abort(403, 'Token CSRF invalide')

        return func(*args, **kwargs)
    return wrapper
```

# Module 6

## Headers de Sécurité Complets

# Configuration Nginx Complète

```
server {
    listen 443 ssl http2;
    server_name exemple.com;

    # Configuration SSL/TLS
    ssl_certificate /etc/ssl/certs/exemple.com.crt;
    ssl_certificate_key /etc/ssl/private/exemple.com.key;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256;
    ssl_prefer_server_ciphers on;

    # Headers de sécurité
    add_header X-Frame-Options "DENY" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-XSS-Protection "1; mode=block" always;
    add_header Referrer-Policy "strict-origin-when-cross-origin" always;
    add_header Permissions-Policy "geolocation=(), microphone=()" always;
}
```

# HSTS et Preload

```
# HSTS - Force HTTPS pendant la durée spécifiée
# max-age : durée en secondes (2 ans recommandé)
# includeSubDomains : applique aussi aux sous-domaines
# preload : permet l'inclusion dans les listes des navigateurs
add_header Strict-Transport-Security
    "max-age=63072000; includeSubDomains; preload" always;
```

## Processus de preload

1. Configurer HSTS avec le flag preload
2. Soumettre le domaine sur hstspreload.org
3. Le domaine sera intégré dans Chrome, Firefox, etc.
4. Attention : difficile à annuler une fois preloadé

# CORS - Cross-Origin Resource Sharing

```python
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)

# Configuration CORS restrictive
CORS(app, resources={
    # Appliquer uniquement aux routes /api/*
    r"/api/*": {
        # Domaines autorisés à appeler l'API
        "origins": ["https://frontend.exemple.com"],

        # Méthodes HTTP autorisées
        "methods": ["GET", "POST", "PUT", "DELETE"],

        # Headers personnalisés autorisés
        "allow_headers": ["Content-Type", "Authorization"],

        # Autoriser l'envoi de cookies cross-origin
        "supports_credentials": True
    }
})
```

# Module 7

## Sécurité des API

# Authentification API - JWT

```python
import jwt
from datetime import datetime, timedelta

SECRET_KEY = 'clé-secrète-très-longue'
ALGORITHM = 'HS256'

def create_token(user_id, role):
    """Crée un token JWT pour l'utilisateur."""
    payload = {
        'sub': user_id,           # Subject : identifiant utilisateur
        'role': role,             # Role pour les autorisations
        'iat': datetime.utcnow(), # Issued At : date de création
        'exp': datetime.utcnow() + timedelta(hours=1)  # Expiration
    }

    # Signature du token avec la clé secrète
    token = jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)
    return token
```

# Validation JWT

```python
def validate_token(token):
    """
    Valide un token JWT et retourne les données.
    Lève une exception si le token est invalide ou expiré.
    """
    try:
        # decode vérifie la signature et l'expiration
        payload = jwt.decode(
            token,
            SECRET_KEY,
            algorithms=[ALGORITHM]
        )
        return payload

    except jwt.ExpiredSignatureError:
        raise ValueError("Token expiré")

    except jwt.InvalidTokenError:
        raise ValueError("Token invalide")
```

# Middleware d'Authentification API

```python
from functools import wraps
from flask import request, jsonify

def require_auth(func):
    """Décorateur exigeant un token JWT valide."""
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Récupérer le header Authorization
        auth_header = request.headers.get('Authorization')

        if not auth_header or not auth_header.startswith('Bearer '):
            return jsonify({'error': 'Token manquant'}), 401

        # Extraire le token (après "Bearer ")
        token = auth_header[7:]

        try:
            payload = validate_token(token)
            # Stocker les infos utilisateur pour la route
            request.user = payload
        except ValueError as e:
            return jsonify({'error': str(e)}), 401

        return func(*args, **kwargs)
    return wrapper
```

# Rate Limiting pour API

```python
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(
    app,
    key_func=get_remote_address,  # Limite par IP
    default_limits=["100 per hour"]  # Limite par défaut
)

@app.route('/api/search')
@limiter.limit("10 per minute")  # Limite spécifique
def api_search():
    # Cette route est limitée à 10 requetes par minute par IP
    return jsonify(results=search())

@app.route('/api/login')
@limiter.limit("5 per minute")  # Plus restrictif pour le login
def api_login():
    # Limite les tentatives de connexion
    return jsonify(token=authenticate())
```

# Travaux Pratiques - Sécurisation

## Exercice 1 : Protéger une application vulnérable

1. Prendre le script vulnérable fourni
2. Corriger l'injection SQL avec des requetes préparées
3. Ajouter l'échappement XSS
4. Implémenter la protection CSRF

## Exercice 2 : Configurer les headers

1. Analyser les headers actuels avec curl

# Synthèse Jour 4

| Protection | Technique |
|---|---|
| Injection SQL | Requetes préparées, ORM |
| XSS | Echappement, CSP |
| CSRF | Tokens, SameSite cookies |
| Authentification | JWT, sessions sécurisées |
| Autorisation | RBAC, vérification serveur |
| Transport | HTTPS, HSTS |

# Préparation Jour 5

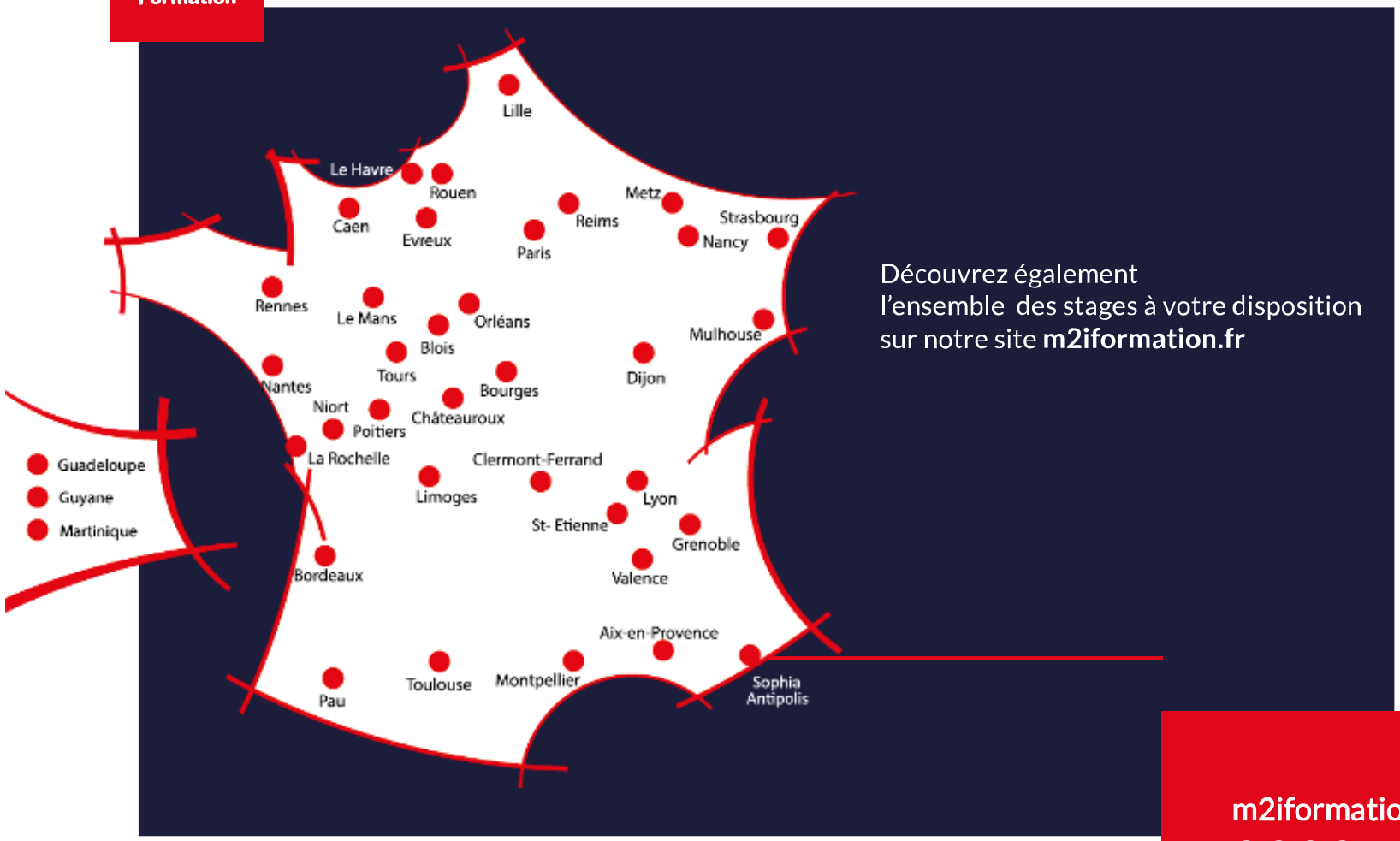
## Au programme demain

- Automatisation SAST et DAST
- Intégration dans les pipelines CI/CD
- Durcissement système et serveur
- IDS/IPS et monitoring

# Questions

## Ressources

- OWASP Cheat Sheet Series : cheatsheetseries.owasp.org
- Security Headers : securityheaders.com
- Mozilla Observatory : observatory.mozilla.org




Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr