Sécurité et API Rest

la sécurité : un problème évolutif

- attaques et vulnérabilités en évolution constante
- nécessite d'adapter les parades de manière permanente
- Niveau de sécurité d'une application = niveau de sécurité de son maillon le plus faible

- Quelques particularités dans le cas des applications basées sur des services/API

OWASP Top 10 Web App Security Risks 10.2021

1. violation du contrôle d'accès
2. exposition de données sensibles – défaut cryptographique
3. injection
4. Conception non sûre
5. Mauvaise config sécurité
6. utilisation de composants avec des vulnérabilités
7. authentification/session
8. Intégrité des données
9. Logging & monitoring insuffisant
10. falsification de requête inter-site (SSRF-CSRF)

OWASP Top 10 **API** Security Risks 2019

1. défaut de contrôle d'accès à 1 ressource
2. violation d'authentification de l'utilisateur
3. exposition excessive de données
4. manque de limitations de requêtes
5. défaut de contrôle d'accès à une fonctionnalité
6. Affectations de données non contrôlées
7. Mauvaise config sécurité
8. injections
9. Mauvaise gestion des items exposés
10. Logging & monitoring insuffisant

Les principes de base

- Les **échanges** entre le client et le serveur ne sont **pas sûrs** : il faut envisager la possibilité qu'ils soient détournés
- Ne **jamais faire confiance** aux données fournies par **l'utilisateur**
  - Query, body, Cookie, Headers
  - Fichiers uploadés
- Attention à **l'authentification**/autorisation/contrôle d'accès
- Utiliser des techniques **éprouvées et à jour**

Les pratiques de base

- Valable pour les applications et les API
- Utiliser **https** pour protéger les échanges entre client et serveur
- **Valider** les données en entrée sur le backend
  - **toutes** les données (url, body, cookie, headers)
  - présence, type, taille, schéma
  - Utiliser une librairie
    - validate.js, express-validator
    - respect/validation, SlimValidation

- **Filtrer** et/ou nettoyer les données en entrées pour éliminer les **injections**
  - Injection SQL, NoSQL, XSS
  - **toutes** les données (url, query string, body, cookies)
  - **utiliser exclusivement des requêtes préparées lors de l'accès au sgbd**

- Fichiers uploadés : ATTENTION
  - .html, .php, exécutables …
  - **NE JAMAIS LES AFFICHER ou LES EXECUTER**
  - Les stocker de manière inaccessible à une url
  - Les bannir si possible

En production :

- **limiter les messages d'erreur** à des messages génériques :
  - Pas d'information sur la nature de l'erreur
  - Pas d'information sur le code ayant provoqué l'erreur
  - Pas d'information sur la cause de l'erreur

- **tracer** (logger) tous les comportements inhabituels
  - Erreurs diverses
  - Echecs d'authentification, de validation de données, d'autorisation/contrôle d'accès …
- **Analyser les logs** pour détecter les attaques répétées ou au long cours

En développement et production :

- Utiliser des composants mis à jour régulièrement
- **Faire les mises à jour en prod**

- En dev : utiliser un fichier de dépendances (composer.json/package.json) qui permet des mises à jour
  - Éviter les require trop stricts exigeant 1 version précise
- Utiliser un security checker de configuration
  - Exemple : https://security.symfony.com/ pour conposer.lock

- **Faire attention à la configuration globale du système**
  - permissions mal configurées
  - comptes par défaut ou de test toujours présents
  - erreurs conduisant à l'exposition d'informations sensibles (stack trace, noms d'objets dans la BD …)
- les problèmes apparaissent souvent :
  - à la transition dev → prod
  - lors de mises à jours en prod
- pour faire face :
  - minimiser la plateforme d'exécution, nettoyer les composants inutiles
  - définir un processus de validation/audit automatisé pour les mise en production

Faire spécialement attention aux données sensibles

- Identifiants, mots de passe, données personnelles
- adresses, données de paiement

- **transport en https**

- **mots de passe :** stockage haché-salé avec un algorithme de qualité cryptographique

- **autres données sensibles** : stockage cryptées avec un algo à jour et des clés de bonne qualité
  - attention aux décryptage automatique

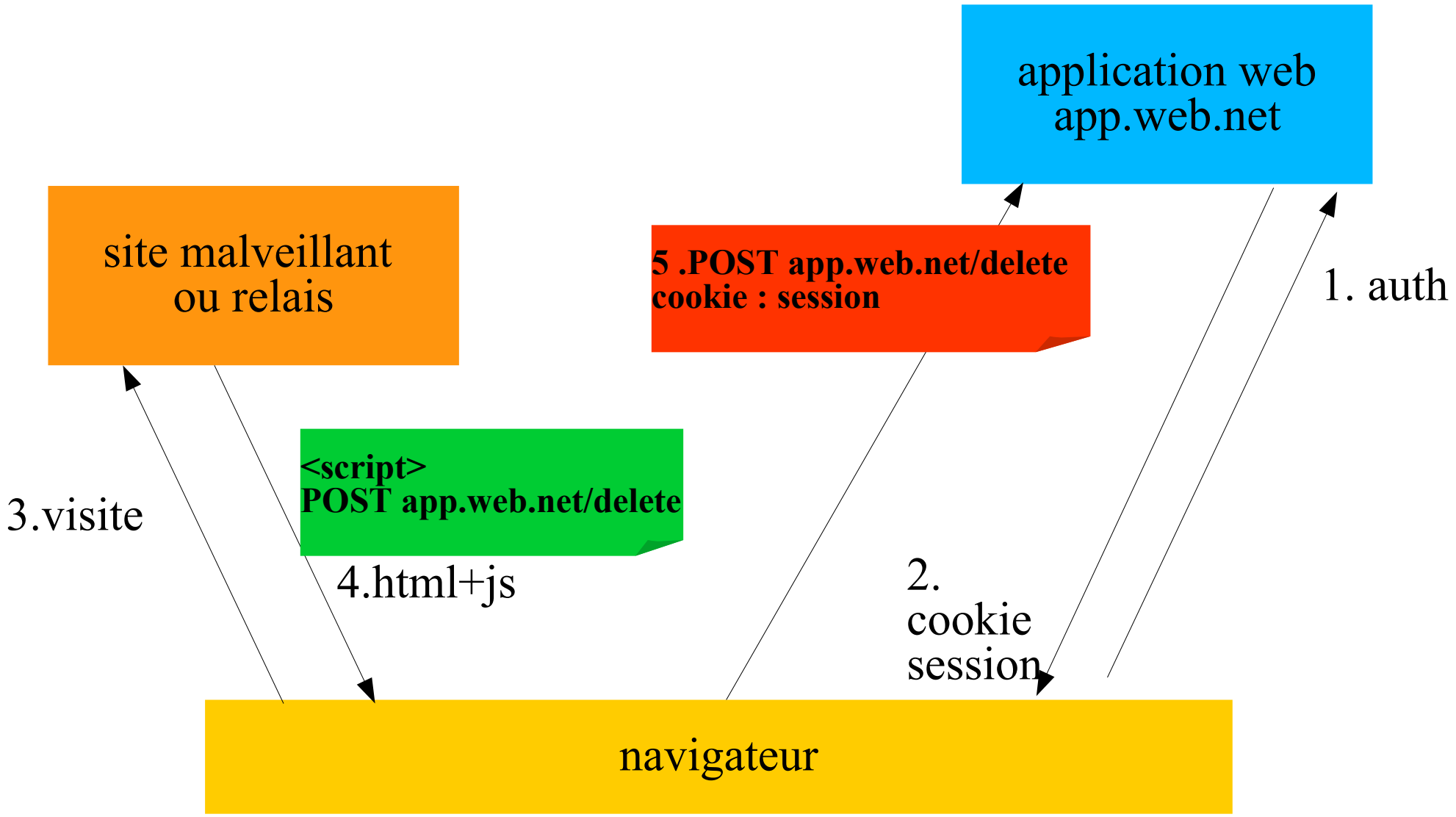
- **Attention aux ressources sensibles accessibles directement sans contrôle d'accès**
  - Exemple : /api/conf/db.ini

- distinguer les ressources publiques des autres
  - répertoire public → vhost ou deny all par défaut
  - contrôle d'accès dans les méthodes des contrôleurs

10. Cross-Site Request Forgeries / CSRF

- **Principe** : faire un sorte qu'un internaute disposant d'une session sur une application exécute **à son insu** une action/requête sur cette application en forgeant cette requête et en la rendant accessible au travers d'un lien, d'un script js, d'une faille XSS ou d'une balise image

CSRF : principe



d'où vient le problème ?

- le problème est du au fait que l'identifiant de session ou d'autorisation est transporté dans 1 cookie, donc de manière transparente et automatique

comment résister dans 1 API

- Ne **JAMAIS** utiliser de session
    - utiliser des token d'authentification type JWT
- Ne **JAMAIS** transporter les token d'authentification dans un cookie

Comment résister sans une appli

- respecter la sémantique GET/POST
  - seuls les formulaires/POST sont à protéger
- Garantir que les requêtes sont issues de l'interface de **votre** application uniquement

- Principe :
  - contrôler le header `Origin:`
  - Ajouter un jeton (token) caché aléatoire dans les formulaires de l'application,
  - Le jeton est stocké en session
  - Reçu et comparé lors du traitement des requêtes et/ou des formulaires

comment résister

- faire les contrôles/ajout de champ de manière générique :

- lors de la génération des formulaires :
  - générer et insérer un token CSRF
  - le stocker en session
- en préambule de toutes les requêtes de type POST
  - vérifier la présence du jeton et sa concordance avec le jeton stocké en session

avec Slim

- Mettre la génération de token et le contrôle de token dans 1 middleware

- le coder soit-même en utilisant une librairie :
  - CSRF Guard PHP
  - PHP SecurityMultiTool

- Utiliser un middleware existant :
  - https://github.com/slimphp/Slim-Csrf
  - https://github.com/oscarotero/psr7-middlewares

autre solution pour résister dans une appli

- Ne plus utiliser les mécanismes de session utilisant un cookie (PHPSESSID, JSESSIONID)

- Les remplacer par l'utilisation d'un token JWT

OWASP Top 10 **API** Security Risks 2019

1. **Défaut de contrôle d'accès à 1 ressource**
2. **violation d'authentification de l'utilisateur**
3. **exposition excessive de données**
4. **Manque de limitations de requêtes**
5. **défaut de contrôle d'accès à une fonctionnalité**
6. **Affectations de données non contrôlées**
7. Mauvaise config sécurité
8. injections
9. Mauvaise gestion des items exposés
10. Logging & monitoring insuffisant

3. exposition excessive de données

- Cas d'une API retournant des résultats de requêtes contenant trop de données sans nécessité
- Causes :
  - Implantation trop générique
  - Fonctionnalités automatiques des framework

```
$data = Command::where('id', '=', $id)->with('client')
                ->first()
                ->toJson();

$rs->getBody()->write($data);
return $rs;
```

3. exposition excessive de données

- Prévention :
  - Ne jamais confier au client le tri des données, ne lui transmettre que celles dont il a besoin
  - Faire une revue systématique des données retournées par l'API pour vérifier qu'elles ont légitimes et nécessaires aux fonctionnalités et adaptés aux clients potentiels
  - Ne pas utiliser de fonctions génériques de type `toJson()` ou `toString()`
  - Ne pas oublier les réponses en cas d'erreurs

6. Affectations non contrôlées – (mass assignment)

• Les ressources manipulées par l'API peuvent contenir de nombreuses propriétés. Certaines ne doivent pas être modifiées par les clients.

• Dans certains cas le contrôle réalisé par l'API sur des mises à jour de données peut être insuffisant :

```
PUT /api/clients/567437  {"prenom" : "michel","age": 32 }
200 OK

GET /api/clients/567437
200 OK { "nom" : "dupont", "age" : 32, "solde" : 150}

PUT /api/clients/567437 { "age" : 32, "solde" : 150000}
200 OK
```

6. Affectations non contrôlées – (mass assignment)

- Cause : le contrôleur utilise des techniques d'affectation de masse offertes par le framework sans contrôle :

```
$data = $rq->getParsedBody();
$user = Client::firstOrFail( $args['id']);
$user->fill($data);
$user->save();
```

- Comment résister :
  - Éviter l'affectation de masse
  - Valider le schéma des données reçues en entrée
  - Utiliser une liste blanche pour les propriétés autorisée ($fillable en Eloquent)
  - Utiliser une liste noire pour les propriétés interdites ($guarded en Eloquent)

4. Manque de limitations de requêtes

- Certaines requêtes reçues par une API peuvent être consommatrices de ressources d'exécution (stockage, réseau, mémoire, cpu).
- Les ressources nécessaires peuvent être liées aux paramètres fournis par les utilisateurs
- Limites sensibles :
  - Timeout execution
  - Mémoire allouée maximum
  - Nombre de fichiers et de processus
  - Taille des données uploadées
  - Nombre de requêtes par client
  - Taille des pages de liste retournées par 1 requête

4. Manque de limitations de requêtes

- Exemples :

- Une api d'upload de photos créant des vignettes de différentes tailles
  - Un utilisateur upload une image 6000x8000-24bits
  - Et recommence …

- **`GET /api/commands?page=1&size=5000000`**

4. Manque de limitations de requêtes

Comment se protéger :

- Limiter les ressources système
  - C'est facile avec docker
- Définir et contrôler les valeurs limites des paramètres dans les requêtes
- Définir et limiter dans le temps le nombre de requêtes pour chaque client
  - Par exemple : 100 req / heure
  - Facile avec une API Key, sinon utiliser IP
- Définir et contrôler les tailles maximum des données uploadées

OWASP Top 10 **API** Security Risks 2019

1. **défaut de contrôle d'accès à 1 ressource**
2. **violation d'authentification de l'utilisateur**
3. exposition excessive de données
4. Manque de limitations de requêtes
5. **défaut de contrôle d'accès à des fonctionnalités**
6. Affectations de données non contrôlées
7. Mauvaise config sécurité
8. injections
9. Mauvaise gestion des items exposés
10. Logging & monitoring insuffisant

Identification / authentification / autorisation / contrôle d'accès

- Identification : Qui êtes-vous ?
- Authentification : pouvez-vous le prouver ?
- Autorisation : maintenant que je sais qui vous êtes, voici ce que vous pouvez faire et ne pas faire
- Contrôle d'accès : contrôler que vous êtes autorisé à faire ce que vous faites

Rappel : quelques pratiques essentielles

- **HTTPS !**
- Stockage des mots de passe HACHÉS ET SALÉS
  - Comparer les empreintes
- Utiliser des techniques et des algorithmes reconnus et à jour
  - Ne pas réinventer la roue
  - Ne pas programmer ses propres algo
  - Utiliser les algos de hachage et de génération aléatoire les plus forts possibles

authentification/autorisation dans une api

Clé d'API / token opaque :

```
GET /dogs/?apikey=FTSRghxu78hskkx9Nqfr345h7GGFDE21h
```

- Clé d'api : pour authentifier 1 client, à réserver pour des échanges entre serveurs ou appli cliente sûre (pas en js!)
- Risque : partage accidentel de l'url
- Intérêt : facilité de partager 1 ressource
- **Attention** : utiliser une valeur aléatoire générée par un algo de qualité cryptogrqphique

authentification/autorisation dans une api

Session et identifiants de session dans un cookie, token transportés dans des cookie : NE PAS UTILISER dans une API

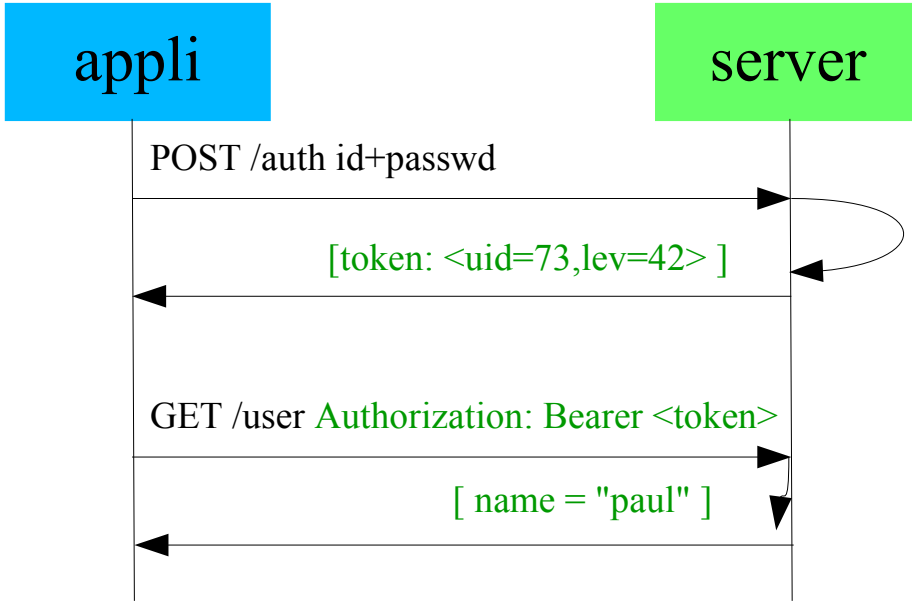
- CSRF
- Passage à l'échelle
- Principe REST : requêtes stateless

authentification/autorisation basée sur un token type JWT

- **principe** : l'authentification retourne un token JWT transparent contenant une autorisation utilisée par les requêtes d'accès aux ressources

```
GET /commandes/345443876
Authorization: Bearer FTSRghxu78hskkx9Nqfr345h7GGFDE21h
```

- A utiliser dans le cas où l'authentification n'est pas partagée entre plusieurs applications/services/back end



authentification/autorisation basée sur un token type OAuth

- A utiliser lorsque l'authentification doit être partagée entre plusieurs services ou backends, ou pour permettre aux utilisateurs de s'authentifier avec un compte sur un service externe

1.Défaut d'autorisation/contrôle d'accès à une ressource

- Le manque de contrôle d'accès à une ressource peut permettre l'accès à une ressource sensible par manipulation de l'identifiant de ressource
  - J'ai une autorisation d'accès sur le compte `labanque.com/account/1337`
  - Je tente le coup : `GET labanque.com/account/`**`1336`**
  - Oh miracle ! Ça marche, j'ai accès au compte de Bill Gates
  - On peut avoir le même problème avec un ID transporté dans un header

1.Défaut d'autorisation/contrôle d'accès à une ressource

- Prévention :
  - Avoir un mécanisme d'autorisation/contrôle d'accès basé sur les utilisateurs
  - Faire les contrôles d'accès dans toutes les méthodes manipulant des objets en vérifiant l'autorisation de l'utilisateur authentifié
  - Utiliser des ID de ressources non prédictibles, par exemple UUID plutôt que auto-increment

2. Violation de l'authentification utilisateur

- compromission de l'authentification permettant à l'attaquant de s'approprier l'identité d'un utilisateur

- **Attaques automatisées, credential stuffing, force brute**
  - authentification multi-facteur
  - limiter ou allonger la durée des tentatives + logger
- **password par défaut, bien connus ou faibles**
  - éviter les credentials par défaut (e.g. root/"")
  - contrôler les mots de passe avec des listes (e.g. danielmiessler/SecLists)
  - longueur et police à jour avec les recommandations

- **mécanisme d'inscription/recouvrement de password**
  - éviter les questions/réponses
  - ne pas distinguer les cas d'erreurs permettant d'obtenir des informations même partielles (identifiant, nom …)
  - Protéger contre la brute force

- **mots de passes et tokens lisibles, cryptés ou hachés/générés avec un algo faible**
  - HTTPS !
  - hachés-salés avec des algos reconnus comme sûrs
  - valeurs aléatoires générés avec des algo sûrs
  - Mots de passes stockés sous forme hachée

5. Défaut de contrôle d'accès à une fonctionnalité

- Cas où il est possible d'envoyer des requêtes légitimes sur un point d'entrée en principe réservées à des utilisateurs particuliers
  - Envoi d'une requête vers un point d'accès privé
  - Envoi d'une requête en utilisant une méthode réservée à certains utilisateurs (POST, PUT …)
  - Envoi d'une requête réservée à un groupe X en appartenant à un groupe Y

  - Exemple : je m'ajoute moi-même dans un groupe admin

```
POST /api/users/56GTR457/groups

{"mail":"mechant@malin.com","role":"admin"}
```

- Autre exemple :

  `GET /api/admin/users` librement accessible
- Prévention :
  - Contrôle d'accès dans toutes les méthodes des contrôleurs
  - Séparer les contrôleurs d'administration, les faire hériter d'un contrôleur abstrait implantant les contrôles d'accès
  - Deny par défaut, grant explicites

conclusion

- Surveiller, logger, mettre à jour, limiter les messages d'erreur
- Utiliser des outils et algos validés (hash, random, crupt ..)
- Attention aux outils des frameworks : affectation massive, toJson() …
- Limiter les ressources consommées
- Filtrer/valider les données pour éviter l'injection
- Mots de passe stockés hachés/salés
- Gérer la force brute
- Tous les échanges en https