Sécurité Applicative

M1 WEB - Secure Coding
Ve. 28 Juin 2019 - PHELIZOT Yvan

```
var b64img = window.location.hash.substr(1);
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    var reader = new FileReader();
    reader.onloadend = function() {
      document.write(`
    <a href="${b64img}" alt="${atob(b64img)}">
     <img src="${reader.result}">
    </a>`);
    }
    reader.readAsDataURL(this.response);
  }
};
xhttp.responseType = 'blob';
xhttp.open("GET", b64img, true);
xhttp.send();
```

Où est la faille?

Comment l'exploiter?

Comment la détecter?

Comment la corriger?

Summary

- Fixing Things Correctly
- Secure Coding for Java
- Secure Coding Web
- Secure by Design Principles

Fixing Things Correctly

Injection

- Using values from unknown source as trusted
  - Validate: test if a value correspond to a format
  - Sanitize: remove dangerous content
  - Escape: securing output
- Not applying best practices
  - SQL: Prepared Statement
- Allowing
  - Excessive rights (user in mysql, injection command not chrooted, …)
  - Boundaries between services
  - XSS/CAPTCHA?

With Prepared Statement (PHP & PDO)

```
$stmt = $conn->prepare(SELECT * FROM User WHERE
userId = ?");

$stmt->bind_param(“s”, 1);

$stmt->execute();
```

Or use ORM (correctly…)

Without Prepared Statement (PHP & PDO)

$res = $conn->query(SELECT * FROM User WHERE userId = ?");

With Prepared Statement

```
$stmt = $conn->prepare(SELECT * FROM User WHERE
userId = ?");

$stmt->bind_param(“s”, 1);

$stmt->execute();
```

Or use ORM (correctly…)

Always validate inputs

Origin — Is the data from a legitimate and/or trusted sender?

Size — Is it reasonably big?

Lexical content — Does it contain the right characters and encoding?

Syntax — Is the format right?

Semantics — Does the data make sense?

Always validate inputs

Strongly typed at all times (oups…)

Length checked and fields length minimized

Range checked if a numeric

Unsigned unless required to be signed

Syntax or grammar should be checked prior to first use or inspection

Validate all inputs

Use Whitelist approach

Blacklist	Whitelist

`if(dir.matches("\n/\\\\\\\."))  {`
`  return;`
`}`
	`if (!dir.matches("\\w+")) {`
	`  return;`
	`}`

Validate on server-side

Should I reject as soon as I can?

Broken Authentication

- **Implementing your own solution**
- Improperly using authentication protocols
  - OAuth2 protocol/Kerberos/SAML
  - MFA
- Using weak authentication
  - MD5 instead of PDKF/B2CRYPT
  - Predictive session IDs
  - Default password
- Disclosing sensitive information
  - Session ID in URL
- Not detecting brute-force attack

Broken Access Control

- Not verifying every access (complete mediation)
  - Predictive ID (/user/1)
  - Predictive URL (/admin)
  - Use of methods (GET, POST, DELETE, PUT, PATCH, OPTIONS, ...)
- Tampering data
- Not limiting rights

Sensitive Data Exposure

- Not/Badly encrypting data
  - Storing/sending sensitive data in clear text
  - Weak cryptographic solutions (md5, base64, …)
- Returning dangerous information
  - Stacktrace
  - Too much information
- Displaying sensitive information
  - Password on profile
  - Secret key on SVC
- Logging sensitive information
  - Don’t log keys, session IDs, ...

Security Misconfiguration

Insecure by default

- Default login/password (admin/admin)
- Security features not activated by default
- Improperly configured
- Unclean installation (Wordpress, …)
- Ignoring security configuration

Using Components with Known Vulnerabilities

- Out-of-date components
- Tools
  - Dependency Check
  - npm audit
  - Clair (Docker)
  - ...

Secure Coding for JavaScript

Javascript secure coding

- Front
  - Validation ⇒ UI =/= Server-side validation
  - Minification =/= Security
- Avoid dangerous functions (**eval**)
- Use up-to-date dependencies
- Use frameworks correctly

Javascript is subtle

- [] == ![]; true == [];, true == ![]; // non transitif
- Math.min() > Math.max() // => true
- Number.MAX_SAFE_INTEGER +1 == Number.MAX_SAFE_INTEGER +2

Javascript - Type Juggling

```
authenticate(dbHash, userHash): String {
  return dbHash == userHase;
}
```

Javascript - Typescript

```
authenticate(dbHash, userHash): String {
  return dbHash === userHase;
}
```

No more problems?

Javascript - Typescript

```
authenticate(dbHash, userHash): String {
  return dbHash === userHase;
}
```

No more problems?

Secure Coding for Web

CSRF

- Exécution d’une requête falsifiée
- Solution: génération d’un nonce pour empêcher l’attaque (“CSRF Token”)
- Express: npm i csurf
- Angular: cookie X-XSRF-TOKEN

Headers for Security

Set-Cookie httpOnly	Empêche le cookie d'être lisible depuis javascript
Set-Cookie secure	Indique au navigateur que le cookie ne doit être transmis que lors d'un échange sécurisé (HTTPS)
X-XSS-Protection	Protection basique contre le XSS, à la charge du navigateur
Access-Control-*	Définit les entêtes CORS
X-Content-Type-Options	Empêche le navigateur de déterminer le type du fichier
expect-ct	Demande aux navigateurs de vérifier la chaîne de confiance des certificats
Strict-Transport-Security	HTTP Strict Transport Security
X-Frame-Options	Configurer le type d'iframes à inclure pour éviter le clickjacking
Content-Security-Policy	Politique de sécurité générale pour les contenus. Prévient un grand nombre d'attaques

API Security

- Authentication ⇒ Authorization
- Secure communication : HTTPS
- Components : API Gateway, Service Mesh
- IDOR, Injection, Security Misconfiguration ...
- HTTP Response code 500
- getMessage content in response
- Security by obscurity: hidden Path
- Swagger, api-doc

CORS

- API: SOP
- On ne peut accéder qu’aux ressources de même origine
- Autoriser
  - GET/POST/HEAD
  - Content-Type: application/x-www-form-urlencoded, multipart/form-data ou text/plain
- Requête “pre-flight”

CORS

- Access-Control-Request-Method: POST
- Access-Control-Allow-Origin: <un adresse>, *
- Access-Control-Allow-Headers: Authorization, ...
- Access-Control-Allow-Credentials: true

OAuth

- Access delegation: users to grant to their information without giving them their credentials
- =/= authentification (OIDC)
- SPA, implicit flow
- Token in session/local storage
- Alternative: iframe, OAuth PIXE, Not using OAuth
- DDoS: Quota, limitations,

OAuth2/Open ID Connect

- OAuth2: Délégation de services
- OIDC: Authentification (basé sur OAuth2)
- ***Démo avec Spring!***

Secure by Design principles

Secure by design principles

1 Minimize attack surface area

2 Establish secure defaults

3 Principle of Least privilege

4 Defense in depth

5 Fail securely

6 Don’t trust services

7 Separation of duties

8 Avoid security by obscurity

9 Keep security simple

10 Fix security issues correctly

TP

Projet JS

$ npm init

$ npm install express body-parser helmet express-session accesscontrol passport passport-openidconnect cors cookie-parser express-session-fixation --save

$ npm audit

$ npm install eslint-plugin-security # bonus

Projet

- Créer une API type CRUD gérant une liste de nourriture (/foods)
- Authentifier un utilisateur avec OIDC (Avec Google API par exemple)
- Autoriser
  - Un utilisateur anonyme à voir tous les types de nourritures
  - Un utilisateur authentifié à ajouter de la nourriture
  - Un administrateur peut tout faire
- Gérer correctement les exceptions
- Activer helmet
- Empêcher les attaques de type “session fixation”
- Autoriser les requêtes de toute origine (CORS)
- Échapper les caractères HTML dans le nom de la nourriture