

Type de vulnérabilité	Nombre	Sévérité	Examen
Stored XSS	3	Critique	Faux positifs , il n'y a aucune donnée utilisateur et pas de script malveillant possible ici. Abordé plus en détail en page 3 de ce document. J'ai quand même ajouté un nettoyage de l'affichage par principe.
Stored Command Injection	1	Critique	Faux positifs, il n'y a pas de vulnérabilité car les fichiers concernés sont créés par l'application avec des noms déjà strictement nettoyés. Abordé plus en détail en page 4 de ce document. Néanmoins par précaution supplémentaire j'ai ajouté escapeshellarg() par dessus.
Client DOM Code Injection from AJAX	6	Haute	Il faudrait que l'attaquant arrive à changer le contenu du retour AJAX envoyé par le serveur... mais admettons. Dans 2 cas c'était destiné à un attribut href : ce contenu est maintenant vérifié. Dans les autres cas c'était un eval() et en effet c'est à éviter autant que possible : j'ai donc modifié la façon de faire, il n'y en a plus.
Relative Path Traversal	5	Haute	Faux positifs pour les 3 premiers cas relevés , c'est nettoyé par Clean::lettres_chiffres() ou Clean::param_chemin(), comme le compte-rendu le relève d'ailleurs. Faux positifs aussi pour les 2 cas suivants , \$_FILES['userfile']['tmp_name'] n'est pas une vulnérabilité, c'est de \$_FILES['userfile']['name'] dont il faut se méfier et c'est déjà le cas.
Dangerous File Inclusion	1	Haute	Faux positif , il n'y a rien de dangereux ici, le nom du dossier est une constante, le nom du fichier est strictement nettoyé, et en plus il y a un format de liste blanche autorisé.
Deserialization of Untrusted Data	1	Haute	La fonction impliquée recuperer_fichier_infos_serializees() n'est pas appelée avec comme paramètre des entrées utilisateur, mais du contenu généré par l'application, il n'y a donc aucun risque. J'ai ajouté en paramètre l'option [allowed_classes' => false] si cela peut rassurer. Je pourrai aussi étudier le remplacement du format de ces transmissions d'informations par du JSON.
Reflected XSS	1	Haute	Nettoyage de \$_COOKIE[COOKIE_STRUCTURE] ajouté à l'endroit mentionné.
SSRF	1	Haute	Faux positif déjà dans le rapport à l'identique au § Relative Path Traversal.
Broken or Risky Hashing Function	32	Moyenne	Faux positifs. Tous les usages de md5() relevés ne servent qu'à associer un identifiant provisoire unique à une image / un texte / une situation, mais en aucun cas à sécuriser un mot de passe ou autre.

Use of Cryptographically Weak PRNG	8	Moyenne	Faux positifs. Les 5 utilisations de mt_rand(), comme au § précédent, ne servent pas ici à sécuriser un mot de passe ou autre. Les 3 utilisations de str_shuffle() servent seulement à élaborer un mot de passe provisoire pour une connexion locale, il n'y a pas non plus de vulnérabilité ici.
HttpOnly Cookie Flag Not Set	4	Moyenne	Faux positifs pour les 2 de class.Cookie.php : il s'agit de cookies secondaires dont certains doivent en plus être accessibles au javascript. Remarque prise en compte pour les 2 de class.Session.php.
Unchecked Input For Loop Condition	2	Moyenne	Faux positifs (confirmés par l'IA) ; j'ai quand même ajouté une limite.
HttpOnly Cookie Flag Not Set	2	Moyenne	Faux positifs , il s'agit de cookies très secondaires devant de surcroît être accessibles au javascript.
Missing HSTS Header	1	Moyenne	Ajouté si https disponible, même si c'est déjà paramétré au niveau du serveur (mais c'est davantage une recommandation qu'une vulnérabilité).
Open Redirect	1	Moyenne	C'est pris en compte, j'ai ajouté une liste blanche.
SSL Verification Bypass	1	Moyenne	Paramétrage volontaire de CURLOPT_SSL_VERIFYPEER à cause de certaines installations... Comme pour « Missing HSTS Header », c'est maintenant paramétrable serveur par serveur selon l'endroit où SAcOche et installé.
Client JQuery Deprecated Symbols	33	Basse	Faux positifs : ce qui est signalé n'est en rien une vulnérabilité et n'est pas du tout déprécié, au pire ce n'est juste pas optimal ; j'ai quand même remplacé au passage les sélecteurs :first et:last par:first-child et:last-child.
Use of Non Cryptographic Random	20	Basse	Faux positifs , de surcroît redondants car déjà répertoriés précédemment : il s'agit encore d'utilisations de mt_rand() et str_shuffle() qui sont parfaitement adaptées à l'usage qui en est fait.
Unsafe Use Of Target blank	3	Basse	Assez ridicule, on commence vraiment à chipoter : attribut rel="noopener noreferrer" omis sur 3 liens parmi... 228. Ajouté sur ces 3 liens.
Use of Insufficiently Random Values	2	Basse	Faux positifs , ces 2 utilisations de Math.random() sont parfaitement adaptées à l'usage qui en est fait.
Insufficient Sanitization for XSS	1	Basse	Redondant avec le tout premier point évoqué, et donc déjà pris en compte.
Client Use Of Iframe Without Sandbox	1	Basse	Ici ce n'est pas le code de SAcOche mais celui de « Highcharts ». Je pense avoir fait le nécessaire pour ajouter une sandbox à cet iframe.

Concernant **la pseudo-vulnérabilité critique « Stored XSS »**, j'ai donné le bout de code à une IA et lui ai demandé ce qu'elle en pensait, pour avoir confirmation.

Voici sa réponse :

Ce signalement est très probablement un **faux positif**, mais je peux expliquer pourquoi Checkmarx le remonte.

Pourquoi c'est signalé

Les outils SAST comme Checkmarx raisonnent en « taint analysis » : ils suivent les flux de données d'une **source** vers un **sink**. Ici :

- **Source** : le résultat de `$BD_handler_root->query($query)` — toute donnée issue d'une base de données est traitée comme potentiellement contaminée (le « Stored » de « Stored XSS » signifie justement : *le payload viendrait de données stockées en base*).
- **Sink** : `echo html(print_r($row, TRUE))` — sortie directe dans le HTML.

Checkmarx ne « comprend » pas ta fonction `html()` : elle s'appuie sur `perso_mb_detect_encoding_utf8()` et `perso_utf8_encode()`, des fonctions maison inconnues de son modèle. Ne pouvant pas prouver que c'est un sanitizer équivalent à `htmlspecialchars()`, il considère le flux comme non nettoyé et remonte la vulnérabilité par prudence.

Pourquoi c'est un faux positif en pratique

1. **Pas de source contrôlable par l'utilisateur** : la requête filtre sur `SACOCHE_STRUCTURE_BD_USER`, une constante PHP, et les colonnes lues (`host`, `user`, `Select_priv`) de `mysql.user` ne sont modifiables que par un administrateur de la base (via `GRANT/CREATE USER`). Un attaquant capable de faire ça aurait déjà un contrôle bien plus direct sur l'application.
2. **Il y a bien un échappement** : `htmlspecialchars($text, ENT_COMPAT, 'UTF-8')` neutralise `<`, `>`, `&` et `"`. La sortie est insérée dans un contexte texte (`<code>...</code>`), pas dans un attribut entre quotes simples, donc l'absence d'échappement de `'` (`ENT_COMPAT` vs `ENT_QUOTES`) est sans conséquence ici.
3. **Même le pire scénario est inoffensif** : si un nom d'utilisateur MySQL contenait `<script>`, il serait affiché échappé.

Seule nuance théorique : sur certaines configurations d'hébergement mutualisé, l'utilisateur du panel pourrait créer des comptes MySQL aux noms arbitraires, et si ce script affichait les entrées de `mysql.user` sans le filtre `WHERE user="..."`, il y aurait un vecteur (stocké en base → affiché à l'admin). Mais avec le filtre sur une constante et l'échappement présent, ce vecteur est fermé.

Que faire

- Signaler un **false positive** dans le rapport Checkmarx avec cette justification (données d'administration, pas de saisie utilisateur ; échappement `htmlspecialchars` effectif).
- Sinon, faire reconnaître `html()` comme un sanitizer : dans Checkmarx, cela se configure via les **data sanitizers / cleansing functions** (CxAudit permet de marquer une fonction comme nettoyante, ou d'utiliser les annotations de suppression).
- Éventuellement simplifier le code pour que l'outil voie `htmlspecialchars` directement (peu recommandé si la fonction gère l'encodage de façon indispensable, comme ton commentaire l'indique).

Donc : ton analyse est correcte, l'outil remonte un flux DB → echo qu'il ne peut pas prouver nettoyé, mais il n'y a pas de vecteur d'attaque réaliste ici.

Concernant **la pseudo-vulnérabilité critique « Stored Command Injection »**.

L'outil Checkmarx alerte sur une utilisation trouvée de `exec()`.

L'usage de cette fonction est en effet à prohiber autant que possible, nous sommes tous d'accord là dessus, c'est la base.

Dans SACoche cela n'apparaît exceptionnellement qu'à un seul endroit (parmi 275 000 lignes de code).

Il s'agit de zipper des fichiers.

Comme indiqué en commentaire dans le fichier concerné, s'il y a beaucoup de fichiers à zipper, on peut dépasser le `max_execution_time`.

Afin d'éviter ce problème, on essaye donc d'appeler la commande `zip` directement (si le serveur l'autorise).

En effet, non seulement ce n'est alors pas du temps CPU compté pour PHP, mais en plus c'est 2 fois plus rapide.

La "vulnérabilité critique" de type "Stored Command Injection" présentée par Checkmarx est due au fait que cela est une faille si un nom de fichier est volontairement mal formé (par exemple contenant des caractères de séparation de commandes), car cela pourrait permettre d'exécuter des commandes sur le serveur.

Sauf qu'ici les fichiers à zipper sont des fichiers créés par l'application avec des noms générés par l'application.

Donc il n'y a pas de vulnérabilité exploitable, même si dans ce cas précis l'alerte de Checkmarx est compréhensible et instructive.

De surcroît, comme on n'est jamais trop prudent, j'ai ajouté `escapeshellarg()` sur chaque nom de fichier.

Ainsi, c'est même désormais doublement sécurisé.

Ce qui n'empêche pas Checkmarx, si on relance une analyse, de maintenir qu'il s'agit d'une vulnérabilité critique...