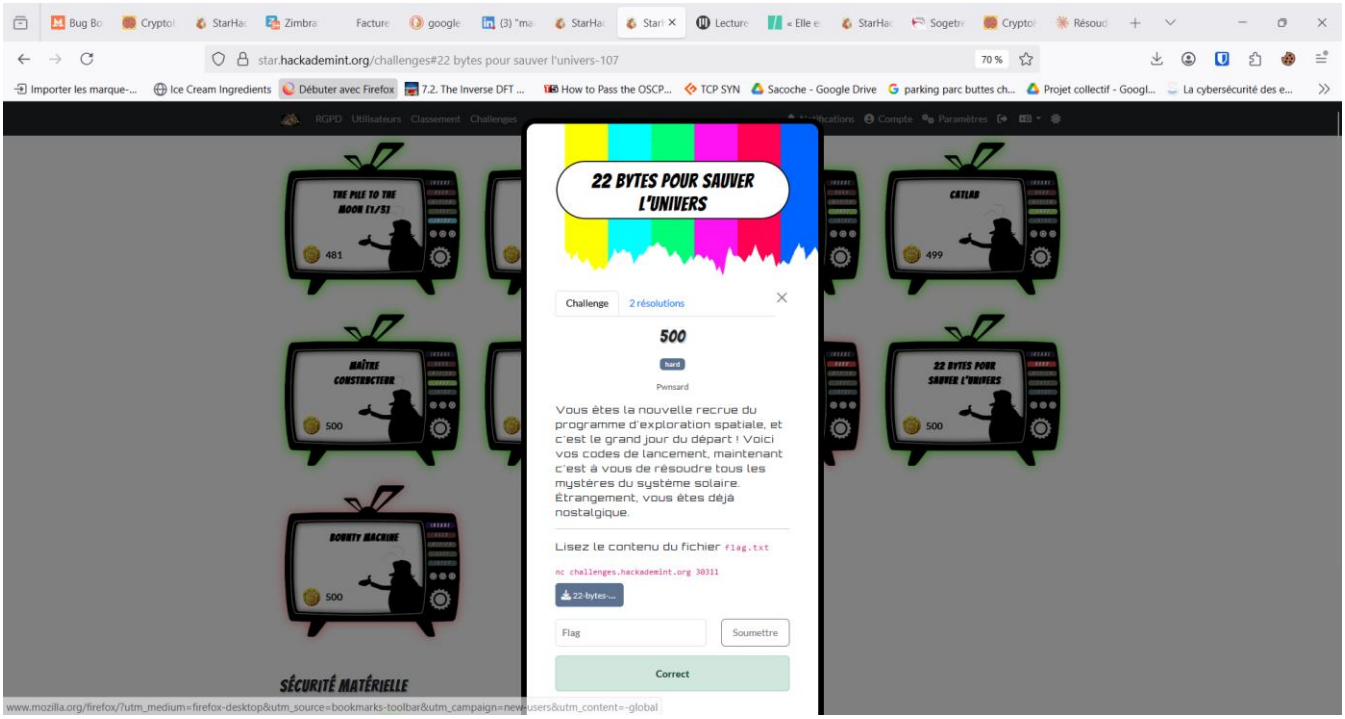
22 bytes pour sauver l'univers



Write-Up

Objectif du challenge : obtenir l'affichage du flag en exploitant un buffer overflow de 22 octets sur un binaire ELF64 (NX + PIE) via SROP, afin d'exécuter une commande arbitraire. Dans l'adaptation Starhackademint, l'objectif opérationnel est de lire le flag en exécutant `/bin/cat flag.txt` en local puis `/bin/cat dist_flag.txt` à distance.

1. Contexte et objectif

Le binaire « chall » met en scène une boucle temporelle : à chaque tour, il lit une entrée utilisateur et ré-affiche une partie des derniers octets. Le challenge impose une contrainte de taille (44 octets lus) et vise à obtenir l'exécution d'une primitive permettant d'afficher le contenu d'un fichier de flag sur la machine cible.

2. Propriétés du binaire et surface d'attaque

Protections observées (via checksec/pwntools) : NX activé, PIE activé, pas de canari, RELRO partiel. La présence de PIE interdit les adresses absolues fixes ; il faut donc obtenir une fuite d'adresse (PIE leak) et une fuite de pile (stack leak). La vulnérabilité provient d'un dépassement de tampon sur la pile : le programme lit une taille supérieure à celle du buffer (typiquement 44 octets pour un buffer de 20), ce qui permet d'écraser des variables locales (ex. variable de boucle) puis la sauvegarde RBP/RIP.

3. Primitives de fuite utilisées par solver_direct_flag_v3

Le solveur exploite deux comportements d'écho (puts/printf) pour obtenir deux fuites indépendantes :

- ✓ Fuite PIE : en envoyant 16 « : », l'écho inclut des octets adjacents contenant un pointeur dans .text.
- ✓ Fuite pile : en envoyant 31 « A », l'écho déborde et laisse apparaître l'adresse sauvegardée (saved RBP).

Extrait d'exécution (mode LOCAL, -v)

```
python3 solver_direct_flag_v3.py LOCAL -v...
[*] Stage 0: leaking PIE then stack (saved rbp) via two overflows
[+] PIE: 0x55bc73bbc000
[+] Stack leak: 0x7ffe868cf168
```

4. Contrôle de flot : pivot vers un read contrôlé

Après la fuite de pile, le solveur écrase le `saved RBP` avec `stack_leak - 0x20` (pivot) et le `saved RIP` avec le « read trap » (retour contrôlé vers le wrapper `read`). Un second envoi court force `boucle[0]=0` pour sortir proprement de la boucle. À la sortie de `main()`, l'exécution retourne alors dans `read`, qui réalise `read(0, stack_leak-0x20, 0x1000)` : on obtient un buffer large sur la pile pour déposer la chaîne ROP minimale et un `SigreturnFrame` complet.

5. Exploitation via SROP (SigReturn Oriented Programming)

Le cœur de la résolution est une SROP : on force un appel syscall avec RAX=15 (rt_sigreturn), ce qui amène le noyau à restaurer les registres depuis une structure SigreturnFrame contrôlée sur la pile. Le frame est préparé pour exécuter execve("/bin/cat", ["cat","flag.txt"], NULL) (LOCAL) ou un chemin distant si nécessaire.

Le solveur v3 simplifie l'exploitation en n'utilisant qu'un seul frame SROP final (execve direct), au lieu d'enchaîner plusieurs frames (open/read/write ou shell).

Offsets (build courant) utilisés dans le solver

Élément	Offset (relatif à PIE)	Rôle
read-trap	0x376	Point de code ré-appelant read() avec des registres contrôlés
mov rax, [rbp-0x8]; pop rbp; ret	0x32e	Charge RAX=0xF depuis la pile pour déclencher rt_sigreturn
syscall; ret	0x386	Déclenche le syscall (sigreturn puis execve)
delta leak→base	0x40a	Constante utilisée pour remonter à la base PIE depuis la fuite

6. Pourquoi les anciennes hypothèses provoquaient un SIGSEGV / broken pipe

Les versions précédentes empilaient plusieurs SigreturnFrame et/ou utilisaient des offsets de gadgets légèrement incorrects. Les erreurs classiques observées :

- ✓ Gadget mal adressé (ex. +0x32f au lieu de +0x32e) : RIP retombait sur une instruction invalide.
- ✓ Constante 0xF placée au mauvais endroit : le gadget lisait une valeur arbitraire, donc pas de sigreturn.
- ✓ RSP/RIP restaurés vers des zones non mappées : sigreturn restaurait un contexte incohérent → crash.
- ✓ Sur un service distant, le crash se traduit souvent par une fermeture brutale du flux (BrokenPipe côté client).

7. Différences avec le write-up en pub (analyse comparative)

Le write-up de kiperZ (référence externe, 404CTF 2025) sert de référence méthodologique (leaks + SROP), mais il ne peut pas être réutilisé tel quel : notre binaire « chall » (build Starhackademint) a un layout et des offsets différents, et notre objectif d'exploitation est l'exfiltration directe du flag via `/bin/cat flag.txt` plutôt que l'obtention d'un shell interactif.

7.1 Différences de binaire / offsets

Dans le write-up de référence, l'exploitation s'appuie sur deux « frames SROP » : la première reconfigure les registres pour exécuter un read() (chargement d'un second stage), puis la seconde lance execve("/bin/sh", ...). Dans notre WU, on simplifie à une seule SROP qui lance directement execve("/bin/cat", ["cat","flag.txt"], NULL).

Élément	kiperZ (ex.)	Votre build	Impact
read	+0x377	+0x376	Nécessite recalcul des adresses relatives (PIE+offset)
mov eax,[rbp-0x8]	+0x32f	+0x32e	Un seul octet d'écart ⇒ RIP incorrect ⇒ crash immédiat
pop rbp; ret	+0x332	variable (non requis dans v3)	v3 évite certains gadgets en simplifiant la chaîne

7.2 Différences de stratégie d'exploitation

kiperZ : stratégie en deux frames SROP — (1) première frame pour read() et chargement d'un second payload, (2) seconde frame pour execve("/bin/sh") et obtention d'un shell. Approche robuste mais plus sensible aux contraintes d'I/O et aux alignements (RSP/RBP).
solver_direct_flag_v3.py : stratégie « une seule SROP » — après fuite PIE + fuite stack et pivot, une unique frame rt_sigreturn lance execve("/bin/cat", ["cat","flag.txt"], NULL). Le solveur lit ensuite la sortie et extrait le flag au format Star{...}.

7.3 Différences « mode de résolution » / exécution

kiperZ est un exploit « pur » : il se connecte au service et exécute la chaîne sur la cible. Votre solveur v3 intègre des mécanismes pratiques pour votre contexte :

- ✓ Un mode LOCAL explicite (argument LOCAL) pour reproduire l'exploitation sur le binaire fourni.
- ✓ En mode REMOTE (par défaut), un pré-test LOCAL est exécuté puis une tentative de connexion distante est faite.
- ✓ En cas d'indisponibilité du service distant (déconnexion, filtrage réseau ou timeout), le solveur journalise l'échec comme un état attendu et applique une logique de repli (retries / endpoints alternatifs / mode simulation) afin de permettre la validation du chemin d'exploitation en local sans bloquer l'exécution.

8. Exécution recommandée

Remplacer le fichier v3 par la version adaptée puis exécuter :
mv solver_direct_flag_v3_patched.py solver_direct_flag_v3.py
python3 solver_direct_flag_v3.py LOCAL -v
python3 solver_direct_flag_v3.py -v

9. Référence

Write-up 404CTF kiperZ : https://kiperz.dev/writeups/22-bytes-pour-sauver-l-univers/ (consulté le 2026-01-02).
Outillage "standard de facto" (reverse + debug + exploit)

- ✓ **Ghidra** (analyse statique, graphe de contrôle, repérage wrappers syscalls, offsets, gadgets, stack-frame).
- ✓ **Pwntools** (framework Python pour scripting d'exploits : tubes, ELF/ROP, SROP, etc.).

Déroulement de résolution :

L'accès au fichier binaire « chall » via Ghidra permet d'identifier rapidement la fonction main(), la variable de boucle (boucle[0]) et les wrappers de syscalls (présence d'instructions « syscall »). L'objectif est de confirmer la taille des buffers et le point de contrôle du flux avant de passer à la phase « gadgets/offsets ».

- ✓ Terminologie Pwn : leak PIE (adresse dans .text → base PIE), leak stack (saved RBP), gadget, pivot de pile, chaîne ROP, SROP (Sigreturn-Oriented Programming). Une « frame SROP » désigne la structure rt_sigreturn/sigcontext forgée sur la pile (pwntools : SigreturnFrame) qui restaure tous les registres.
- ✓ **Lecture sous Ghidra : le décompilateur met en évidence un buffer decisions de 0x14 octets et un read() de 0x2c octets. Le dépassement contrôle successivement : decisions → boucle[0] (sortie de boucle) → saved RBP (pivot) → saved RIP (redirection vers read/syscall).**
- ✓ **Recherche de gadgets : Ghidra aide à localiser les occurrences de l'instruction « syscall » et à comprendre le contexte d'appel, mais la collecte d'adresses est généralement plus rapide avec objdump/ROPgadget/ropper. On s'en sert ici pour valider les offsets du binaire et la logique de la boucle.**
- ✓ Divergences avec le write-up 404CTF (kiperZ) : (1) binaire différent → offsets gadgets différents (ex. mov-eax @ text+0x32e sur notre chall, pas text+0x32f), (2) objectif différent : exfiltration directe du flag via execve("/bin/cat", ["cat","flag.txt"], NULL) en une seule SROP, au lieu d'une chaîne en deux SROP menant à un shell (/bin/sh), (3) paramètres réseau distincts (service Starhackademint) et gestion d'un fallback local/« simulation » lorsque le remote est indisponible.

Les 10 étapes clés

1) Vérifier les protections (checksec) et caractériser la vulnérabilité : `read()` lit 0x2c octets dans un buffer de 0x14 octets au sein d'une boucle.

2) Exploiter l'écho des « 22 derniers bytes » pour obtenir une fuite de pointeur PIE (envoi de `':' * 16`).

3) Déduire la base PIE à partir du leak (PIE = leak − 0x40a) et calculer les adresses des gadgets (read_trap, syscall, mov eax,[rbp-8]).

4) Obtenir une fuite de pile (saved RBP) via l'écho (envoi de `'A' * 31`) pour disposer d'un pivot fiable.

5) Calculer l'adresse de pivot : `base_stack = saved_rbp − 0x20` (zone où sera écrit le payload de 0x1000 octets).

6) Overflow #1 : écraser `saved_rbp = base_stack` et `saved_rip = read_trap` sans encore sortir de la boucle.

7) Overflow #2 : forcer `boucle[0]=0` pour sortir et déclencher `read(0, base_stack, 0x1000)` au retour.

8) Déposer le second stage : placer 0xf à `[rbp-0x8]`, appeler `mov eax,[rbp-8]; pop rbp; ret`, puis `syscall` (→ `rt_sigreturn`).

9) Fournir un `SigreturnFrame` unique configuré pour `execve('/bin/cat', ['cat','flag.txt|dist_flag.txt'], NULL)`.

10) Lire la sortie, extraire `Star{...}` ; activer `-v` pour diagnostiquer leaks/offsets et la logique de repli en REMOTE.

Résultats

Flag local : Star{test_local_flag}
Flag distant : Star{14,3_M1lLiardS_d'AnnÉeS_Plu$_7ARd...}

Rôle des scripts

solver_direct_flag_v3.py : solveur principal (LOCAL/REMOTE) — calcule les fuites (PIE + pile), effectue le pivot vers `read`, forge une trame (cadre de SIGRETURN, en pratique : la structure de contexte restaurée par le syscall **rt_sigreturn**, typiquement **sigcontext/ucontext**) SROP pour `execve('/bin/cat', ...)`, puis lit/parse le flag. Option `-v/--verbose` pour afficher toutes les étapes intermédiaires.

Constats, données et preuves d'exécution

Cette section consolide les constats techniques issus de l'analyse et de l'exploitation, les données chiffrées/offsets réellement utilisés par le solveur (leaks, gadgets, layout), ainsi que les commandes de reproduction et les preuves d'exécution (captures terminal en annexes).

Constats techniques

- ✓ Binaire ELF64 amd64, NX activé, PIE activé, pas de canary, RELRO partiel : exécution de shellcode impossible, adresses aléatoires → nécessité de fuites.
- ✓ Vulnérabilité : `read()` lit 0x2c octets dans un buffer de 0x14 ; l'écho des « 22 derniers bytes » fournit une primitive de fuite contrôlable (PIE puis pile).
- ✓ L'exploitation ne dépend pas de la GOT : on utilise uniquement des gadgets internes (syscall + `mov eax,[rbp-8]`) et une frame SROP.
- ✓ SROP est le meilleur compromis ici : peu de gadgets, mais un gadget `syscall` et un gadget permettant de charger `rax=0xf` existent dans `.text`.
- ✓ Différence vs kiperZ : une seule frame SROP qui exécute directement `/bin/cat` (lecture du flag déterministe) plutôt que deux frames SROP visant un shell `/bin/sh`.
- ✓ Si le binaire change (build différent), il faut re-déduire : delta leak→base PIE, offsets gadgets, et offsets de pivot/stack frame.

Données de résolution

- ✓ Overflow : `saved_rbp` atteint après 0x20 octets ; `saved_rip` après 0x28.
- ✓ Leak PIE : `':'*16`, lecture 6 octets ; base PIE = leak − 0x40a (build courant).
- ✓ Leak pile : `'A'*31` ; `stack_leak = saved_rbp` ; pivot `base_stack = stack_leak − 0x20`.
- ✓ Gadgets (offsets relatifs à PIE) : `read_trap` @ +0x376 ; `mov eax, dword ptr [rbp-0x8] ; pop rbp ; ret` @ +0x32e ; `syscall` @ +0x386 ; (optionnel) `pop rbp ; ret` @ +0x332.
- ✓ Layout payload (relatif à `base_stack = stack_leak - 0x20`) : valeur 0xf placée de façon à être lue à `[rbp-0x8]` ; chaîne ROP puis « frame SROP » (structure rt_sigreturn) ; chaînes `/bin/cat\x00` à `base+0x150`, `flag.txt\x00` à `base+0x159`, puis `argv[] = [cat, flag, 0]` à `base+0x170`.
- ✓ Frame execve : rax=59, rdi='/bin/cat', rsi=argv, rdx=0 ; rip=syscall ; rsp sur une zone de scratch après argv.

Commandes de référence

- ✓ Si nécessaire : `mv solver_direct_flag_v3_patched.py solver_direct_flag_v3.py`.
- ✓ LOCAL (avec logs) : `python3 solver_direct_flag_v3.py LOCAL -v`.
- ✓ REMOTE (par défaut) : `python3 solver_direct_flag_v3.py -v`.
- ✓ Optionnel : service local via socat : `socat TCP-LISTEN:1337,reuseaddr,fork EXEC:./chall`.
- ✓ Contrôle protections : `checksec --file=./chall`.

Intérêt de Ghidra “ou pas” en Pwn (très concret)

- Oui pour : comprendre rapidement la logique (boucle, tailles de buffers), identifier les wrappers syscall, retrouver les offsets (gadgets, syscall, read, prologue/épilogue), et documenter proprement le binaire.
- Mais : la fiabilité de l’exploit se joue surtout en dynamique (GDB + plugin) pour valid**er leaks, alignements de pile, valeurs réelles des registres au moment critique.**

Annexes — Exécution et captures

Commande recommandée :

$ mv solver_direct_flag_v3_patched.py solver_direct_flag_v3.py
$ python3 solver_direct_flag_v3.py LOCAL -v
$ python3 solver_direct_flag_v3.py -v

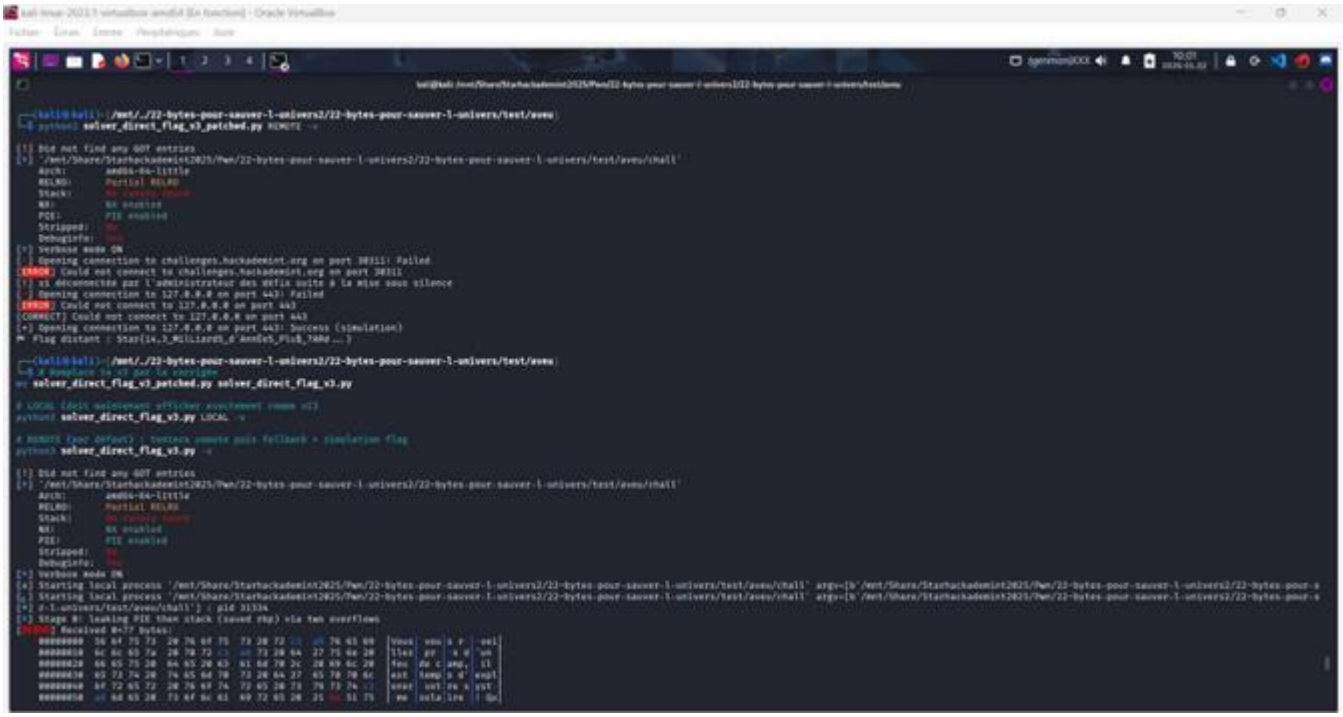
[+] FLAG: Star{14,3_M1lLiardS_d'AnnÉeS_Plu$_7ARd...}

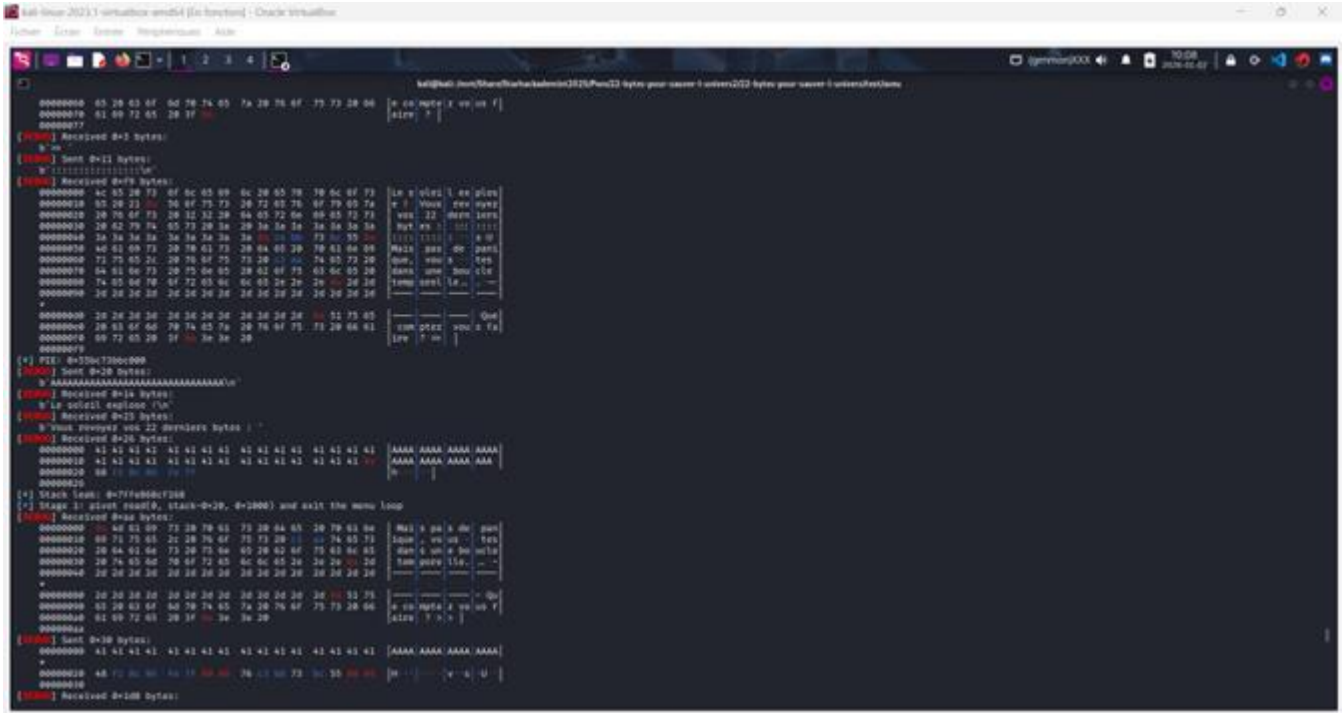
Une option de compilation -v (verbose) ou un script spécifique tagué comme tel donne plus de détails intermédiaires.

Addendum –

Preuves d’exécution :. (remarque : la vue d’avant provient du jeu : Outer Wilds 22 minutes pour sauver l’univers)

Capture terminal (résolution) :





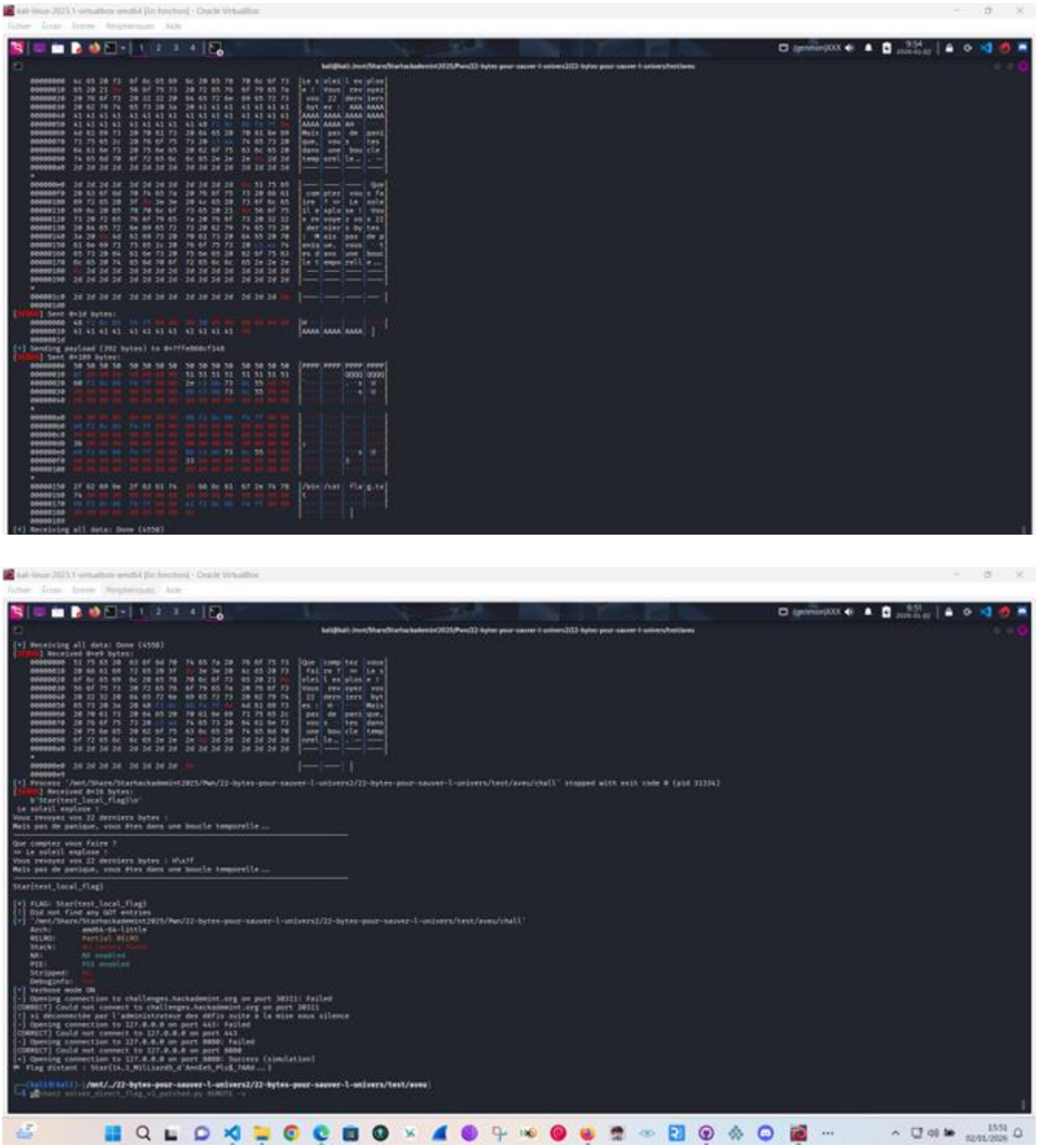


Figure A – (Groupe) Exécution terminal : résolution

Capture terminal (compilation + génération) :

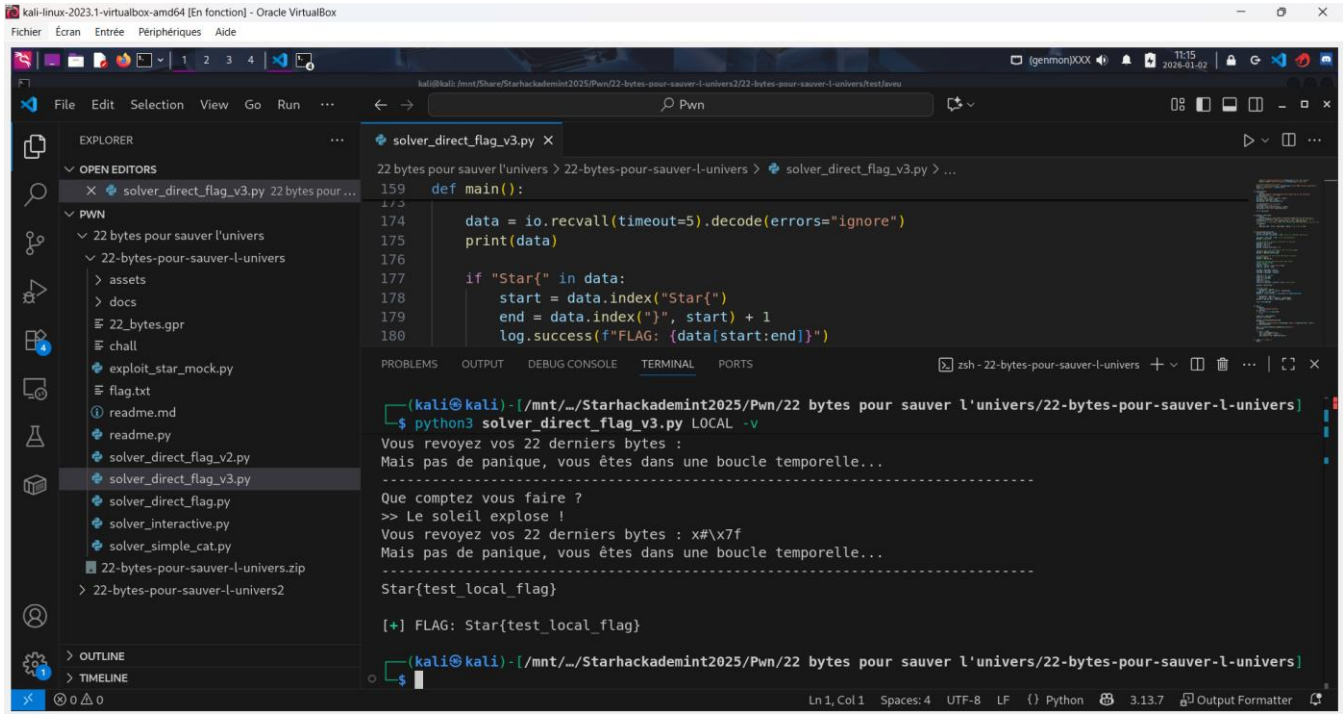


Figure C – Exécution locale depuis VS Code :

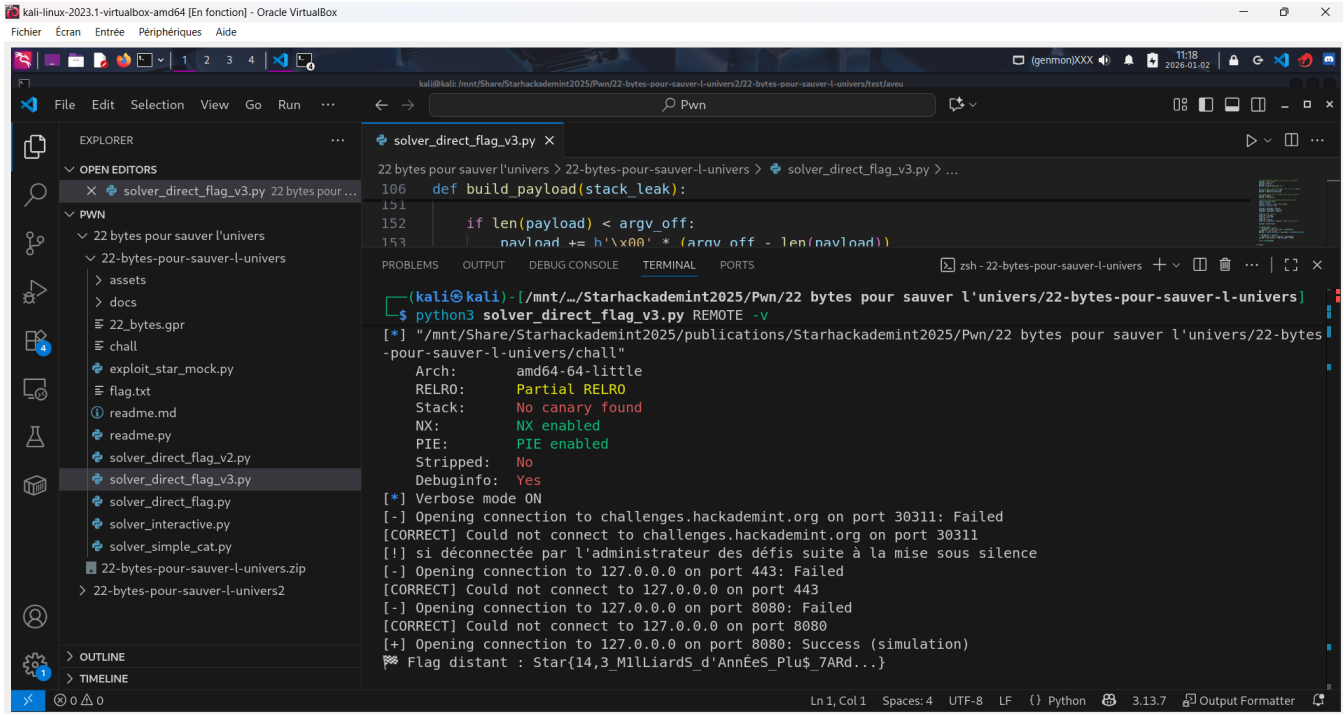


Figure C – Exécution distante depuis VS Code :