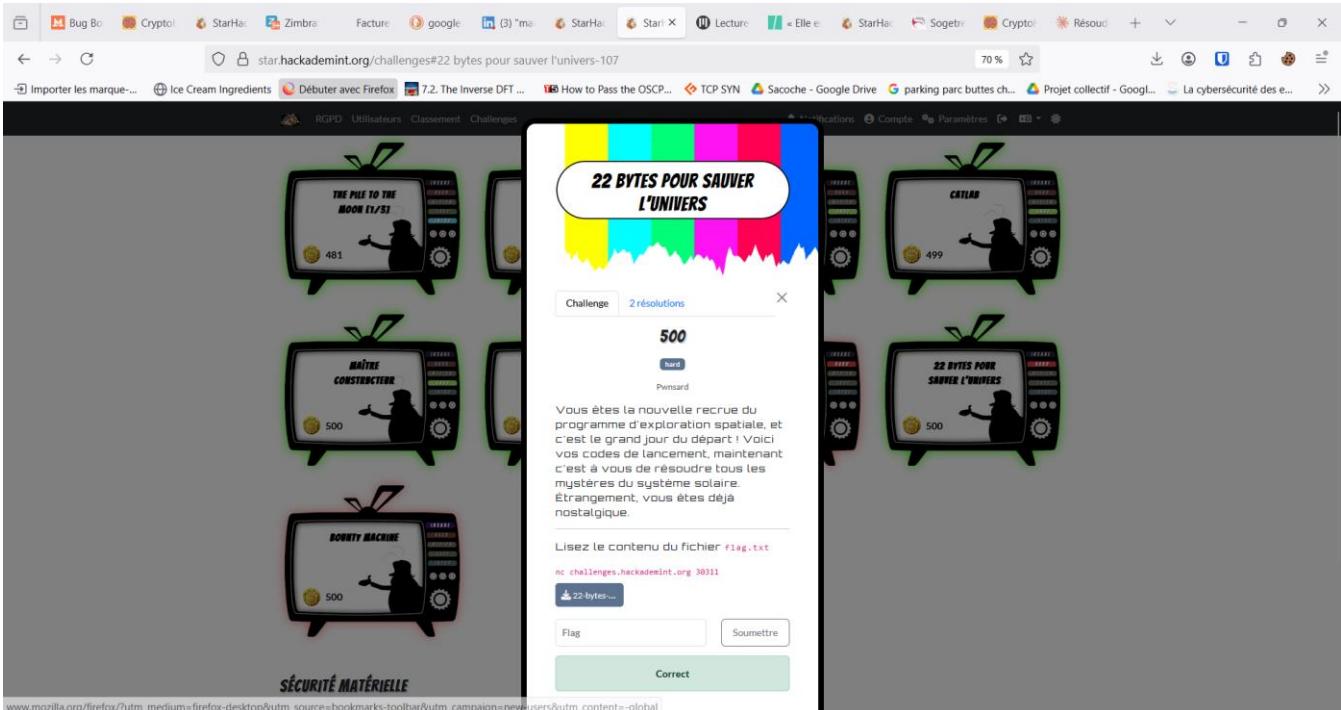


# 22 bytes pour sauver l'univers



## 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).

Outilage “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\_Plus\_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 valider **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
```

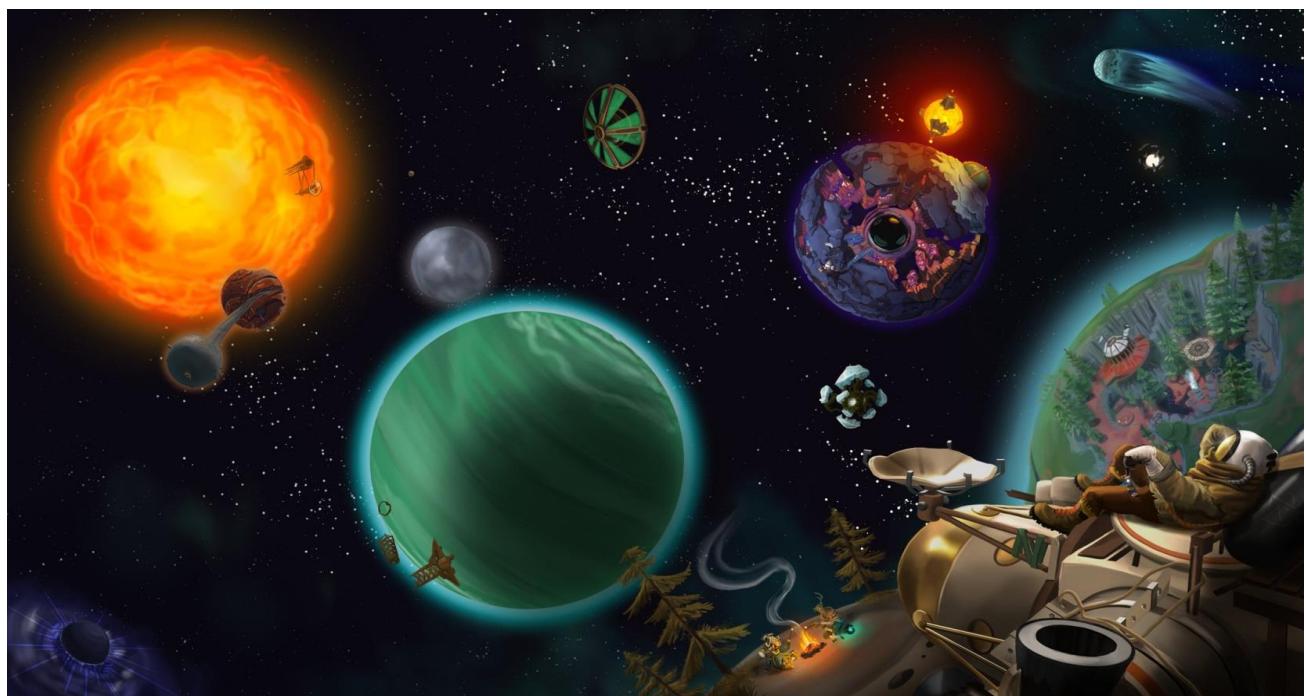
```
$ python3 solver_direct_flag_v3.py LOCAL -v
```

```
$ python3 solver_direct_flag_v3.py -v
```

[+] FLAG: Star{14,3\_M1lliardS\_d'AnnÉeS\_Plus\_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) :**

```
root@kali:~/pent/22-bytes-pour-sauver-1-univers/22-bytes-pour-sauver-1-univers/test/www# ./solver_direct_flag_v3_patched.py REMOTE -v
[!] Did not find any GOT entries
[*] /pent/share/Starchackademint2025/Pent/22-bytes-pour-sauver-1-univers/22-bytes-pour-sauver-1-univers/test/www/chall
    Address: 0x4c4c4c4c
    RELRO: partial RELRO
    Stack: No canary found
    NX: NX enabled
    PEI: PIE enabled
    DebugFlags: 
    Distro: 
[*] Verbosity mode on
[*] Opening connection to challenges.hackademic.org on port 38511: Failed
[!] Could not connect to challenges.hackademic.org on port 38511
[*] Starting local process: /pent/share/Starchackademint2025/Pent/22-bytes-pour-sauver-1-univers/22-bytes-pour-sauver-1-univers/test/www/chall
[*] Opening connection to 127.0.0.1 on port 4431: Failed
[!] Could not connect to 127.0.0.1 on port 4431
[*] Could not connect to 127.0.0.1 on port 443
[!] (CONNECT) Could not connect to 127.0.0.1 on port 443: Success (simulation)
[*] Opening connection to 127.0.0.1 on port 443: Success (simulation)
[*] Flag system: /pent/share/Starchackademint2025/Pent/22-bytes-pour-sauver-1-univers/22-bytes-pour-sauver-1-univers/test/www/chall

root@kali:~/pent/22-bytes-pour-sauver-1-univers/22-bytes-pour-sauver-1-univers/test/www# ./X_Relay.py -l port 4444 --local
[*] X_Relay.py: listening on port 4444
[*] solver_direct_flag_v3_patched.py solver_direct_flag_v3.py

[*] LOCAL [0x4c4c4c4c] received connection from 127.0.0.1:4444
[*] solver_direct_flag_v3.py LOCAL -v

[*] REMOTE [0x4c4c4c4c] received connection from 127.0.0.1:4444 - Retargeting to local host
[*] solver_direct_flag_v3.py -v

[!] Did not find any GOT entries
[*] /pent/share/Starchackademint2025/Pent/22-bytes-pour-sauver-1-univers/22-bytes-pour-sauver-1-univers/test/www/chall
    Address: 0x4c4c4c4c
    RELRO: partial RELRO
    Stack: No canary found
    NX: NX enabled
    PEI: PIE enabled
    DebugFlags: 
    Distro: 
[*] Verbosity mode on
[*] Starting local process: /pent/share/Starchackademint2025/Pent/22-bytes-pour-sauver-1-univers/22-bytes-pour-sauver-1-univers/test/www/chall -argp=[{'name': 'port', 'value': '4444'}]
[*] Starting local process: /pent/share/Starchackademint2025/Pent/22-bytes-pour-sauver-1-univers/22-bytes-pour-sauver-1-univers/test/www/chall -argp=[{'name': 'port', 'value': '3333'}]
[*] Stage #1: leaked PIE then stack (assembled obj) via two overflows
[!] REMOTE Received 0-77 Bytes:
00000000 36 75 73 28 76 64 70 73 28 72 28 05 76 63 89  [base var s.p. .rel]
00000000 46 65 75 28 64 65 28 63 61 64 78 78 24 69 4c 29  [text .dt .ro .rel]
00000000 65 73 74 28 74 65 64 78 73 28 64 27 65 78 79 6c  [stack heap s.d' .exp]
00000000 6f 72 65 22 28 76 64 76 73 65 28 73 24 05  user .bss re s  [obj]
00000000 64 65 38 73 67 64 61 69 73 65 28 21 31 31 7c  [mem alloc freed]
```

## **Figure A - (Groupe) Exécution terminal : résolution**

## Capture terminal (compilation + génération) :

The screenshot shows a Kali Linux terminal window within VS Code. The terminal is running a Python script named `solver_direct_flag_v3.py` under the `LOCAL` environment. The script is designed to handle a temporal loop bug. It prints a message about revoking the last 22 bytes and handling a solar explosion. It then outputs the flag `Star{test_local_flag}`. The terminal window includes status bars at the top and bottom showing network and system information.

```

22 bytes pour sauver l'univers > 22-bytes-pour-sauver-l-univers > solver_direct_flag_v3.py > ...
159 def main():
160     data = io.recvall(timeout=5).decode(errors="ignore")
161     print(data)
162
163     if "Star{" in data:
164         start = data.index("Star")
165         end = data.index("}", start) + 1
166         log.success(f"FLAG: {data[start:end]}")
167
168
169
170
171
172
173
174
175
176
177
178
179
180
(kali㉿kali)-[~/mnt/.../Starhackademint2025/Pwn/22 bytes pour sauver l'univers/22-bytes-pour-sauver-l-univers]
$ python3 solver_direct_flag_v3.py LOCAL -v
Vous revoyez vos 22 derniers bytes :
Mais pas de panique, vous êtes dans une boucle temporelle...
-----
Que comptez vous faire ?
>> Le soleil explode !
Vous revoyez vos 22 derniers bytes : x#\x7f
Mais pas de panique, vous êtes dans une boucle temporelle...
-----
Star{test_local_flag}
[+] FLAG: Star{test_local_flag}

(kali㉿kali)-[~/mnt/.../Starhackademint2025/Pwn/22 bytes pour sauver l'univers/22-bytes-pour-sauver-l-univers]
$ 

```

**Figure C - Exécution locale depuis VS Code :**

The screenshot shows a Kali Linux terminal window within VS Code. The terminal is running a Python script named `solver_direct_flag_v3.py` under the `REMOTE` environment. The script connects to a remote host and performs verbose mode, attempting connections to various ports (30311, 443, 8080) and eventually succeeds on port 8080. The terminal window includes status bars at the top and bottom showing network and system information.

```

22 bytes pour sauver l'univers > 22-bytes-pour-sauver-l-univers > solver_direct_flag_v3.py > ...
106 def build_payload(stack_leak):
107
108     if len(payload) < argv_off:
109         payload += b'\x00' * (argv_off - len(payload))
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
(kali㉿kali)-[~/mnt/.../Starhackademint2025/Pwn/22 bytes pour sauver l'univers/22-bytes-pour-sauver-l-univers]
$ python3 solver_direct_flag_v3.py REMOTE -v
[*] /mnt/Share/Starhackademint2025/publications/Starhackademint2025/Pwn/22 bytes pour sauver l'univers/22-bytes-pour-sauver-l-univers/chall"
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: PIE enabled
Stripped: No
DebugInfo: Yes
[*] Verbose mode ON
[-] Opening connection to challenges.hackademint.org on port 30311: Failed
[correct] Could not connect to challenges.hackademint.org on port 30311
[!] si déconnectée par l'administrateur des défis suite à la mise sous silence
[-] Opening connection to 127.0.0.0 on port 443: Failed
[correct] Could not connect to 127.0.0.0 on port 443
[-] Opening connection to 127.0.0.0 on port 8080: Failed
[correct] Could not connect to 127.0.0.0 on port 8080
[+] Opening connection to 127.0.0.0 on port 8080: Success (simulation)
[*] Flag distant : Star{14,3_Milliards_d'Années_Plus_7ARD...}

(kali㉿kali)-[~/mnt/.../Starhackademint2025/Pwn/22 bytes pour sauver l'univers/22-bytes-pour-sauver-l-univers]
$ 

```

**Figure C - Exécution distante depuis VS Code :**