2023/2024

Shellcode - Rapport

ING 3 CS Groupe B

DASTUGUE Quentin

I. Introduction

Le projet consistait à injecter du shellcode dans un fichier elf. Pour cela, il fallait “rentrer” dans le fichier elf et trouver une manière de changer un PT_NOTE en PT_LOAD. Il fallait ensuite de ce nouveau PT_LOAD pour injecter son shellcode. J’avoue ne pas être bon en assembleur mais j’ai tout de même essayé malgré mon manque d’expérience et de notions. J’ai pu néanmoins apprendre beaucoup grâce à ce projet.

II. Notion et ce que j’ai compris : Lecture ELF

Déjà avant de commencer j’ai réalisé en code en C qui change dynamiquement le pt_note d’un fichier elf, passé en argument, en pt_load. J’ai fais ça car généralement cela m’aide pour définir la structure et la logique du code en assembleur. J’ai fait étape par étape ce code en C. Néanmoins, ce code ne m’a pas vraiment aidé, car certaines notions, comme passer de statique à dynamique sont plus dur en assembleur.

1. La structure d’un fichier ELF :

Comme mon code en C j’ai voulu faire étape par étape. J’ai d’abord commencé par la lecture et s’assurer que le fichier est bien un elf. Pour cette étape, j’ai compris que le format ELF est un standard utilisé sous Linux pour les exécutables, donc un simple code c “Hello World” produit un fichier ELF. Ce format commence toujours par une **signature magique** constituée des 4 octets “`0x7F`”, `'E'`, `'L'`, `'F'`. Cette signature est essentielle pour reconnaître qu’un fichier est bien au format ELF. Pour cela, j’ai utilisé la commande *“readelf -h”* pour examiner les en-têtes elf et identifier les valeurs importantes. Dans mon code, j’ai extrait les 4 premiers octets du fichier grâce au syscall *“read”* pour vérifier cette signature.

2. Assembleur en général :

J’ai mieux compris comment les appels système fonctionnent en assembleur et comment communiquer avec le “noyau” pour effectuer des opérations comme lire, écrire, ou manipuler des fichiers.

Par exemple, pour ouvrir un fichier avec le **syscall open**, le registre **rax** doit contenir la valeur 2 (numéro du syscall `open`), le registre **rdi** reçoit le chemin du fichier, **rsi** indique les options d’ouverture (comme lecture seule `0` ou lecture/écriture 2), et **rdx** peut définir des permissions supplémentaires si nécessaire.

J'ai également vu l'importance de gérer les erreurs pour éviter les plantages du programme. Après chaque syscall, je vérifie si une erreur s’est produite en testant le registre **rax,** si **`rax`** est négatif, cela signifie qu’il y a eu une erreur. Dans ce cas, j’utilise **`js`** (jump if sign) pour afficher un message d’erreur approprié.

3. Parti comparaison

J’ai compris que le “*buffer”* est une zone mémoire réservée pour stocker temporairement des données. Dans mon programme, j’ai utilisé “`resb 4`” pour réserver 4 octets, car seuls les 4 premiers octets du fichier sont nécessaires pour vérifier la signature ELF.

Une fois les données lues, je les ai comparées à la signature attendue pour déterminer si le fichier est valide ou non.

Pour comparer, j'ai découvert l'instruction "***repe cmpsb***" qui permet de comparer automatiquement deux chaînes de données octet par octet. Elle est particulièrement utile pour vérifier si le buffer contenant les 4 octets lus correspond à la signature ELF

III. Problèmes que j'ai rencontré

L'un des premiers blocages que j'ai rencontrés concerne la gestion des arguments pour permettre de passer un fichier en ligne de commande. Initialement, je voulais utiliser les arguments du programme pour spécifier dynamiquement le fichier ELF à analyser. Cependant, je n'ai pas réussi à récupérer correctement ces arguments en assembleur. Cela m'a obligé à coder le chemin du fichier de manière statique dans la section .data.

Pour résoudre cela, j'ai essayé de récupérer l'adresse des arguments via la pile **rsp**, qui est censée contenir les arguments passés au programme. Par exemple, en utilisant **[rsp + 8]** pour accéder au premier argument. C'est un code tout simple comme par exemple:
mov rax, [rsp + 8]
Malheureusement ça n'a pas fonctionné et j'ignore pourquoi, je n'ai donc pas pu gérer correctement les chaînes de caractères.

Une autre difficulté a été la comparaison des octets pour vérifier la signature ELF. Initialement, je voulais utiliser une boucle pour comparer manuellement chaque octet, comme ceci :

```
mov al, byte [rsi]
cmp al, byte [rdi]
jne erreur_signature
inc rsi
inc rdi
loop boucle_comparaison
```

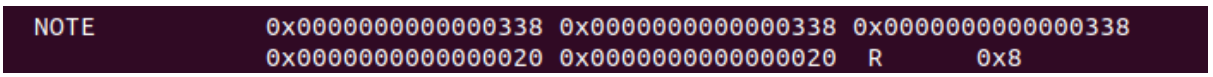
Mais ce code ne marchait pas, j'ai cherché sur internet et j'ai trouvé la manière que j'utilise avec la CMPS instruction. (voir docu)

IV. PT_Note en PT_Load

Ici je vais faire les deux parties, les problèmes et ce que j'ai appris.

1. Trouver la bonne position

Mon premier problème je l'ai rencontré assez vite. J'ai mal compris les valeurs indiqué dans readelf -l hello_world



En fait je savais qu'il me fallait la position de mon premier pt_note, or j'ai pris la valeur que l'on voit ici 0x0338 comme valeur pour l'offset position. En utilisant gdb je me rends compte que je ne suis pas du tout à la bonne position.

En me renseignant plus je me rends compte que la vraie valeur de la position de mon pt_note se trouvait dans le hexedit du fichier (docu). En même temps, j'apprends que les p_flags des headers sont souvent situées à +4 octets de l'offset position, cette information est importante pour plus tard. (docu)

A ce niveau, j'ai fait une méthode pas recommandée, j'ai cherché manuellement l'offset du pt_note dans le hexedit. Pour cela, j'ai modifié la position de l'offset dans mon code avec chaque offset qui avait des valeurs 04, valeur qui correspond a un pt_note. Je savais également que les segments pt_note et pt_load se trouvent au début de mon hexedit. Par exemple : l'offset : 0000002C et 000000B0 (je ne mets pas les screens les lignes sont bien trop longue).

J'ai remarqué ensuite qu' avec 000000B0 (176 en décimal) le flag du premier pt_load a été changé en RWE, cela est dû au +4 octets que je fais pour toucher les p_flags. C'était une très bonne nouvelle, j'avais au moins confirmation que mon code marchait.

```
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x00000000000005f8 0x00000000000005f8  RWE    0x1000
```

Maintenant je devais me déplacer à la bonne position. Je savais que normalement la taille standard d'un header d'un ELF64 est de 56 octets. A partir de là, j'ai essayé de faire 176 +56 octets pour voir si je pouvais toucher le prochain pt_load et j'ai su que j'étais sur la bonne voie :

```
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x00000000000005f8 0x00000000000005f8  RWE    0x1000
LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
               0x00000000000001f5 0x00000000000001f5  RWE    0x1000
LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
```

J'en ai donc déterminé ainsi la valeur du offset position de mon premier pt_note : 456. Avec cette valeur juste, j'ai pu changer mon pt_note en pt_load.

2. Modification d'un segment plus dynamique :

Ensuite j'ai modifié mon code pour qu'il soit un peu moins statique, je pars de la position de début 0xB0 et je boucle en faisant +56 octet jusqu'à ce que je trouve un segment avec un p_type = 0x04. Pour chaque itération, je lis un segment dans un buffer temporaire (dans mon code "*buffer_segment"*) et je compare le type du segment.

J'ai donc appris à comment se repositionner dans le fichier pour modifier son type et ses flags associés. J'ai ainsi compris que pour écrire directement dans un fichier binaire, il suffit de se déplacer à l'offset voulu avec "*lseek"* et d'écrire la nouvelle valeur avec "*write*".

V. Documentation

https://www.tutorialspoint.com/assembly_programming/assembly_cmps_instruction.htm

En C :
https://www.linkedin.com/pulse/elf-files-how-handle-them-c-tariq-abu-elhamd

https://stackoverflow.com/questions/34960383/how-read-elf-header-in-c

https://stackoverflow.com/questions/36246606/correct-way-to-read-elf-files-in-c

Très important :
http://wiki.osdev.org/ELF_Tutorial

Code inspiré
En c : https://github.com/TheCodeArtist/elf-parser/blob/master/elf-parser.c

Docu intéresante sur le sujet du projet :
https://tmpout.sh/1/2.html
https://www.guitmz.com/linux-midrashim-elf-virus/

Bien comprendre ELF

https://cpu.land/becoming-an-elf-lord

https://www.youtube.com/watch?v=bdNCcYddYEU

https://wiki.osdev.org/ELF

https://linuxhint.com/understanding_elf_file_format/

Très important :
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Pour assembleur :

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

https://www.tutorialspoint.com/assembly_programming/index.htm