ING3 – Cybersécurité Groupe B – Sécurité Shellcode
YOUSSOUF ALI Adel
CY Tech
17/12/2024

Projet d'infecteur ELF : compte-rendu

Table des matières

Introduction

Contexte

Les fichiers **ELF** *(Executable and Linkable Format*) sont un format standard pour les fichiers exécutables, les bibliothèques d'objets, les fichiers de débogage et les fichiers de noyau dans les systèmes d'exploitation de type Unix, y compris Linux.
Un fichier **ELF** est structuré en plusieurs segments et sections, chacun ayant un rôle spécifique.

Les segments sont des unités de mémoire qui sont chargées en mémoire lors de l'exécution du programme. L'en-tête de programme décrit ces segments, qui peuvent contenir du code exécutable, des données, ou d'autres informations nécessaires au programme. Les segments permettent une gestion efficace de la mémoire et des ressources, en regroupant les données et le code de manière logique et optimisée pour l'exécution.

Dans le cadre de ce projet, on s'intéresse donc à ces segments et notamment en particulier à deux types d'entre eux : les segments « **LOAD »** et les segments « **NOTE »**.

Les segments de type « **LOAD** » sont très importants car ils contiennent les données et le code qui doivent être chargés en mémoire pour que le programme puisse s'exécuter. Ces segments incluent généralement le code exécutable, les données initialisées et non initialisées, ainsi que d'autres informations nécessaires au fonctionnement du programme.

Les segments de type « **NOTE** » quant à eux sont utilisés pour stocker des informations auxiliaires qui ne sont pas directement nécessaires à l'exécution du programme, mais qui peuvent être utiles pour divers outils et systèmes. Ils permettent une gestion plus riche et plus flexible des informations associées au programme, facilitant ainsi le débogage, l'analyse et la compatibilité entre différents environnements.

Objectif

L'objectif de ce projet est d'écrire un infecteur en assembleur permettant d'injecter du code dans un binaire existant en transformant un segment de type "NOTE" en un segment de type "LOAD". Les segments "NOTE" étant normalement utilisés pour stocker des informations auxiliaires qui ne sont pas essentielles à l'exécution du programme, comme des métadonnées ou des annotations, l'idée est de profiter de cette structure pour y insérer du code supplémentaire.

En modifiant ce segment pour qu'il soit reconnu comme un segment "LOAD", le système d'exploitation chargera ce nouveau code en mémoire lors de l'exécution du binaire infecté. Cela permet d'ajouter des fonctionnalités ou de modifier le comportement du binaire. Ce projet est une manière d'explorer les possibilités de manipulation des fichiers ELF et de comprendre comment les segments peuvent être utilisés de manière créative pour étendre les capacités d'un binaire.

Développement

Outils

Le développement de cet infecteur a été réalisé sur une machine virtuelle Ubuntu que je possède sur Oracle VirtualBox. De plus, j'ai utilisé l'IDE **Microsoft Visual Studio Code** pour la rédaction et notamment l'application **Cutter** pour l'analyse de mon binaire.

Procédé

Étant moyennement à l'aise avec le langage assembleur, j'ai dû premièrement me documenter sur les différentes instructions disponibles avec lesquelles je pourrais donc atteindre mon objectif. J'ai par la suite repris les slides de cours qui nous introduisaient notamment sur les headers d'un programme ELF. À partir de ces dernières, je suis parvenu à déterminer quelles étaient les différentes entrées qui allaient être au cœur de ce projet, à savoir **e_entry** du ELF Header et globalement toute la structure **elf64_phdr** qui détaille l'ensemble des caractéristiques des segments.

Cela aura été un bon point de départ mais j'ai dû faire des recherches supplémentaires pour connaître la taille ainsi que les type de données de chacun des champs de cette structure afin de pouvoir les cibler de manière simple dans mon infecteur. Je suis assez rapidement tombé sur les données suivantes :

```
p_type    dd 0    ; 4 octets
p_flags   dd 0    ; 4 octets
p_offset  dq 0    ; 8 octets
p_vaddr   dq 0    ; 8 octets
p_paddr   dq 0    ; 8 octets
p_filesz  dq 0    ; 8 octets
p_memsz   dq 0    ; 8 octets
p_align   dq 0    ; 8 octets
```

Une fois ces informations en ma connaissance, j'ai pu me mettre à manipuler les différents champs avec différentes instructions afin de voir la manière dont mon binaire réagissait et donc enfin commencer à traiter l'écriture de l'infecteur.

Résultats

L'infecteur

L'objectif global du code assembleur que j'ai écrit est d'ajouter un « **payload** » (qui print que le binaire a bien été infecté) dans un fichier ELF existant (un simple binaire qui print « Je suis le binaire à infecter ») en modifiant un segment de type PT_NOTE, pour y insérer un code qui, lorsqu'il est exécuté, affiche un message et prend le contrôle du programme grâce à la redirection du point d'entrée à un nouvel emplacement.

Le résultat attendu est donc que le binaire infecté fonctionne normalement sans afficher d'erreurs mais au lieu d'afficher simplement « Je suis le binaire à infecter », il est censé d'abord afficher le message choisi dans le payload à savoir « Le programme a été infecté, bravo ! » car le nouveau point d'entrée est l'adresse à laquelle le payload est écrit mais à la fin du payload on effectue un saut au point d'entrée original grâce à l'instruction jmp.

On peut le décomposer en plusieurs étapes :

1. **Ouverture et validation du fichier**

   Le programme commence par ouvrir un fichier ELF spécifié ("./binary") en mode lecture/écriture via un appel système (syscall open). Si l'ouverture échoue, un message d'erreur est affiché et le programme se termine. Ensuite, il lit l'en-tête ELF du fichier (64 octets) pour s'assurer qu'il s'agit bien d'un fichier ELF valide. Le contrôle se fait en vérifiant les premiers octets, qui doivent correspondre à la signature 0x7F 'E' 'L' 'F'. Si ce n'est pas un fichier ELF valide, le programme s'arrête.

2. **Recherche du segment « NOTE »**

   Une fois le fichier validé, le programme commence à rechercher les segments du programme en parcourant les Program Headers. Il accède à l'offset du premier Program Header, et en lit les données (56 octets par entrée) jusqu'à ce qu'il trouve un segment de type « NOTE ». La vérification est faite **en comparant le p_type à la valeur 4** (stockée dans .data) qui est celle par défaut des segments « NOTE ».

3. **Modification du segment**

   Lorsque le programme trouve un segment « NOTE », il affiche un message indiquant que ce segment a été trouvé, puis il le modifie pour le convertir en un segment de type « NODE ». Ensuite, il met à jour les flags de ce segment pour lui attribuer des permissions de lecture, écriture et exécution. Le programme met également à jour la taille et l'adresse virtuelle du segment pour qu'il puisse contenir le code du payload.

4. **Injection du payload**

   Une fois l'ancien point d'entrée sauvegardé, il est remplacé par un nouvel emplacement défini arbitrairement (mais suffisamment grand pour ne pas écraser de données) dans la section .data (**NEW_ENTRY_POINT**). Ce nouvel emplacement est l'endroit où le payload sera inséré.

5. **Écriture finale**

   Après l'injection du payload, le fichier ELF modifié est réécrit avec les nouvelles données. Cela assure que, lors du lancement de ce fichier ELF, l'exécution débutera à l'adresse où le payload a été écrit.

Résultats

Le binaire est compilé à l'aide de gcc et l'infecteur assemblé avec nasm puis ld.

Voici ce qu'on obtient avant injection :

```
adel@ubfinal:~/infecteur_YOUSSOUFALI_Adel$ ./binary
Je suis le binaire à infecter.
adel@ubfinal:~/infecteur_YOUSSOUFALI_Adel$ readelf -l binary

Type de fichier ELF est DYN (fichier exécutable indépendant de la position)
Point d'entrée 0x1060
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
                 0x0000000000000628 0x0000000000000628  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000175 0x0000000000000175  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x000000000000010c 0x000000000000010c  R      0x1000
  LOAD           0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000258 0x0000000000000260  RW     0x1000
  DYNAMIC        0x0000000000002dc8 0x0000000000003dc8 0x0000000000003dc8
                 0x00000000000001f0 0x00000000000001f0  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   0x0000000000002028 0x0000000000002028 0x0000000000002028
                 0x0000000000000034 0x0000000000000034  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000248 0x0000000000000248  R      0x1
```

```
Adresse du point d'entrée:                0x1060
```

Voici ce qu’on obtient après injection :

```
adel@ubfinal:~/infecteur_YOUSSOUFALI_Adel$ nasm -f elf64 infecteur.asm -o infecteur.o
adel@ubfinal:~/infecteur_YOUSSOUFALI_Adel$ ld infecteur.o -o infecteur
adel@ubfinal:~/infecteur_YOUSSOUFALI_Adel$ ./infecteur
PT_NOTE segment trouvé!
PT_NOTE devenu ptnode!
adel@ubfinal:~/infecteur_YOUSSOUFALI_Adel$ ./binary
Le programme a été infecté, bravo !
Erreur de segmentation (core dumped)
adel@ubfinal:~/infecteur_YOUSSOUFALI_Adel$ readelf -l binary

Type de fichier ELF est DYN (fichier exécutable indépendant de la position)
Point d'entrée 0x5000
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
                 0x0000000000000628 0x0000000000000628  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000175 0x0000000000000175  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x000000000000010c 0x000000000000010c  R      0x1000
  LOAD           0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000258 0x0000000000000260  RW     0x1000
  DYNAMIC        0x0000000000002dc8 0x0000000000003dc8 0x0000000000003dc8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  LOAD           0x0000000000005000 0x0000000000005000 0x0000000000000338
                 0x000000000000005a 0x000000000000005a  RWE    0x1000
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x0000000000002028 0x0000000000002028 0x0000000000002028
                 0x0000000000000034 0x0000000000000034  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000248 0x0000000000000248  R      0x1
```

```
Adresse du point d'entrée:                0x5000
```

```
0005000 01b8 0000 bf00 0001 0000 8d48 1a35 0000
0005010 ba00 0027 0000 050f 8b48 3305 0000 ff00
0005020 b8e0 003c 0000 3148 0fff 4c05 2065 7270
0005030 676f 6172 6d6d 2065 2061 a9c3 c374 20a9
0005040 6e69 6566 7463 a9c3 202c 7262 7661 206f
0005050 0a21 1060 0000 0000 0000
000505a
```

Analyse

On constate que sur le header, le segment « NOTE » a bien été changé en « LOAD » et que tous les changements effectués (flags, offset et adresse virtuelle) ont bien été pris en compte.
On voit également que l'adresse du point d'entrée a bien été modifiée. Enfin, lorsqu'on utilise la commande **hexdump** on voit bel et bien que le payload a été rédigé au point voulu.

Le programme se comporte partiellement comme attendu car on voit le message nous indiquant que le programme a été infecté mais le message de départ a disparu et un message indiquant une erreur de segmentation apparaît. Cela laisse penser qu'un paramètre n'a pas été pris en compte et que le jump au point d'entrée ne nous permet pas de faire fonctionner le programme comme au départ.

En étudiant le binaire infecté sur Cutter, voilà ce qu'on obtient :

```
  ;-- segment.LOAD4:
entry0();
0x00005000      mov     eax, 1
0x00005005      mov     edi, 1
0x0000500a      lea     rsi, [0x0000502b]
0x00005011      mov     edx, 0x27  ; '''
0x00005016      syscall
0x00005018      mov     rax, qword [0x00005052]
0x0000501f      jmp     rax
0x00005021      mov     eax, 0x3c  ; '<'
0x00005026      xor     rdi, rdi
0x00005029      syscall
0x0000502b      and     byte gs:[rax + 0x72], dh
0x00005030      outsd   dx, dword [rsi]
0x00005031      jb      0x5095
0x00005034      insd    dword [rdi], dx
0x00005035      insd    dword [rdi], dx
0x00005036      and     byte gs:[rcx + 0x20], ah
0x0000503a      ret
0x0000503b      test    eax, 0x20a9c374
0x00005040      imul    ebp, dword [rsi + 0x66], 0xc3746365
0x00005047      test    eax, 0x7262202c
0x0000504c      invalid
0x0000504d      jbe     0x50be
0x0000504f      and     byte [rcx], ah
0x00005051      or      ah, byte [rax + 0x10]
0x00005054      add     byte [rax], al
0x00005056      add     byte [rax], al
0x00005058      add     byte [rax], al
0x0000505a      invalid
```

On a la confirmation que le segment est bien considéré comme un LOAD et que l'entrée est réalisée au bon endroit mais le programme ne se comporte pas comme prévu par la suite.

Conclusion

Malgré un début d'implémentation qui semblait prendre le bon chemin avec la détermination et la transformation du segment NOTE en LOAD, je ne suis pas parvenu à aller au bout car l'infecteur ne fonctionne pas précisément comme prévu. Je n'ai pas réussi à déterminer la source de mon erreur à temps afin de la résoudre mais je reste satisfait d'avoir pu me pencher sur cette tâche qui m'a permis de développer mes compétences en développement (notamment en assembleur où je partais de loin) mais également à comprendre davantage la structure et le fonctionnement des fichiers ELF.