MOREAU Antoine

ING3 spécialité cybersécurité groupe B

Année universitaire : 2024-2025

17/12/2024

Enseignant : Maxime BOURY

Rapport Projet Shellcode :

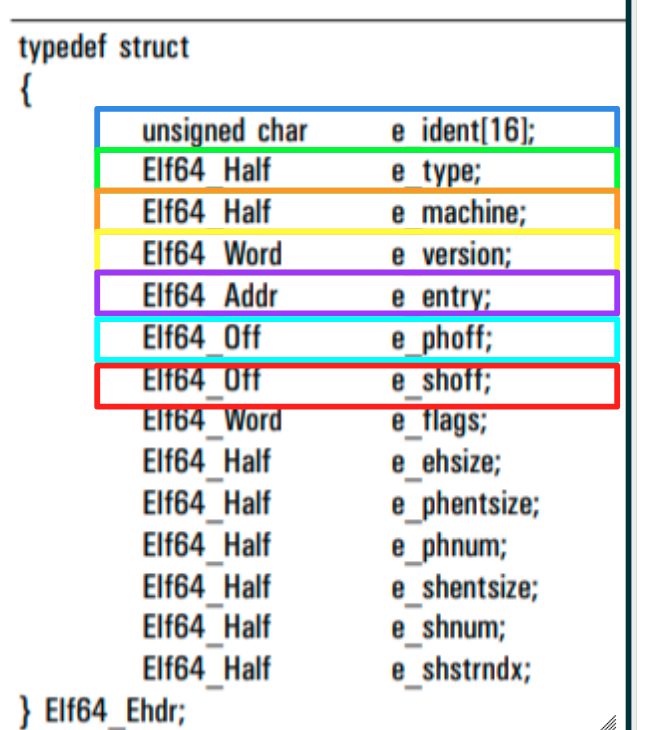
Injecteur PtNote to PtLoad



Notions et objectif :

L’objectif principal du projet est de modifier un fichier de type ELF et d’y injecter du code malveillant. La méthode consiste à modifier un segment PtNote dans les données du binaire à infecter et le changer en un segment exécutable Ptload

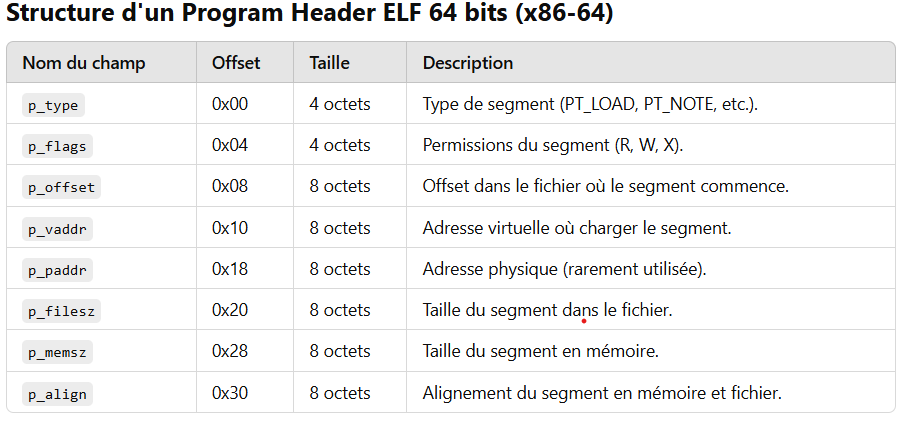
Pour comprendre comment l’on s’y prend il faut d’abord comprendre comment est structuré le header d’un fichier Elf :



Voici la structure et les parties qui nous intéressent le plus sont le type qui définit le type du fichier et nous permet de confirmer avant toute injection que c’est bien un fichier ELF en analysant les 4 premiers octets étant le type dans le header et en les comparant a l’hexadécimal du type d’un fichier ELF. Le deuxième point important est le nombre de headers qui représente les segments contenus dans le fichier ici c’est e\_phnum. En récupérant aussi ce nombre on peut savoir combien il y a de segments à analyser.

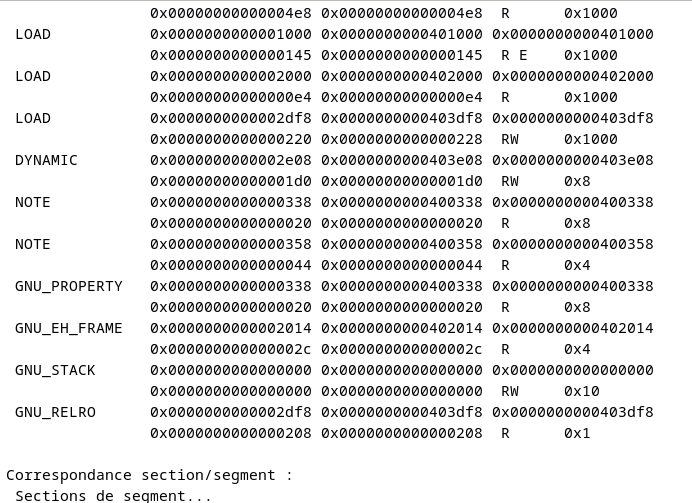
L’un des points de blocage était de ne pas savoir comment accéder à ses données précisément après la lecture et donc pouvoir extraire ces informations. Le type est facile d’accès parce qu’il est au début du fichier mais pour le nombre de header il fallait récupérer cette valeur au bon endroit pendant la lecture. La solution était tout simplement de se renseigner sur la taille de chaque information dans cet header et de se décaler dans l’adressage en conséquence pour lire la bonne valeur. Le décalage pour le type est de 16 octets et pour accéder au champ du nombre de header il faut se décaler de 56 octets.

Une fois qu’on a ce nombre de headers il suffit d’avoir une boucle qui itère parmi ces headers et check leur type pour regarder s’il correspond à celui d’un ptnote, et pour savoir ou trouver le type il faut savoir comment est structuré un programme header :

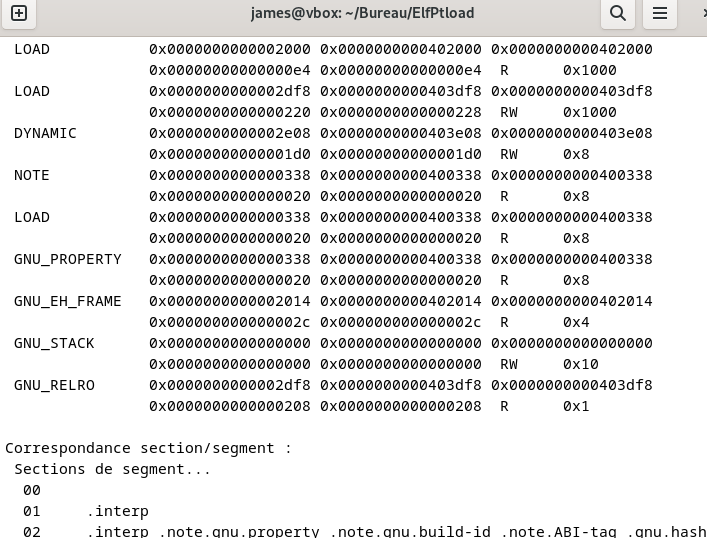


Cela fonctionne pareil que pour la recherche dans le header ELF on regarde la taille de chaque champ et on se décale dans l’adressage pour chercher la valeur qui nous intéresse donc ici le type donc une fois à l’adresse d’un programm header on regarde directement les 4 premiers octets et si le type est 4 c’est que c’est un segment PtNote et on peut donc passer à la suite.

La principale difficulté de cette étape est que je me déplaçais dans le fichier à l’aide du syscall read qui place le curseur à la fin de la partie lue et donc les modifications et la réécriture du buffer dont celle du type en ptload et sa sauvegarde en quelque sorte que ne se réalisais pas au bon endroit et modifiais le segment d’après qui je suppose n’est pas toujours un ptload et donc ne répondait pas à la demande. Dans le readelf cela donne ça :

Avant infection : 

Après infection :

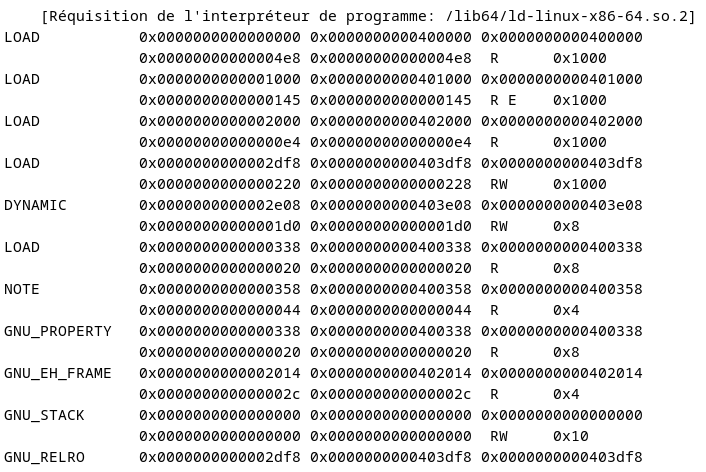


La solution à été d’utiliser le syscall lseek qui permet de se replacer ou l’on veut dans un fichier tant qu’on spécifie l’adresse mais un autre problème est survenu, je me suis rendu compte que je ne lisais jamais toutes les données en même temps dans un buffer et que je réutilisais le même buffer dans ma boucle pour trouver le ptnote et avancer de 56 octets si ce n’en n’était pas un. Le lseek me permet uniquement de me replacer dans des données lues et donc dans un buffer, heureusement au bout d’un moment j’ai compris comment fonctionnait les reads, et le lseek a permis de placer le curseur au premier octet lu dans le buffer ce qui a permis de régler ce problème en écrivant au bon endroit.

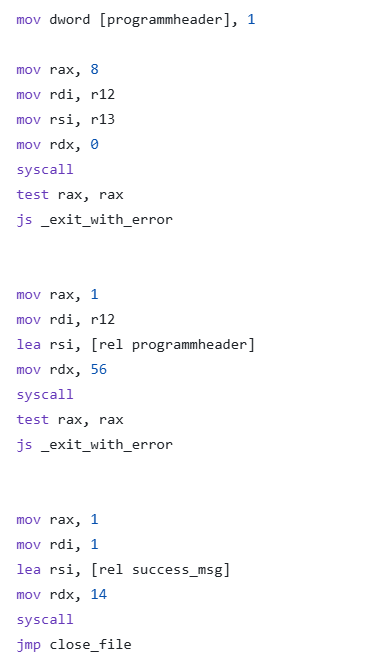
Avant changement :



Après changement :

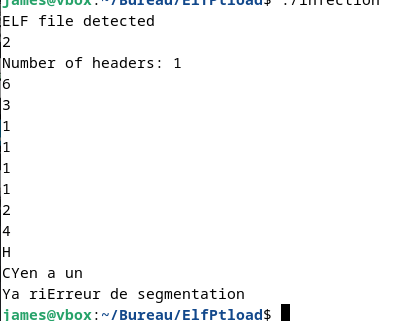


Le premier segment ptnote a bien été changé en ptload et non le segment d’après.



L’étape d’après à consister à résoudre des bugs d’affichage et une segfault persistante a chaque exécution malgré le bon changement et la solution a été très simple les erreurs d’affichage de message de debugg était tout bêtement du a une erreur de taille sur les syscall write ou il faut définir la taille du message et n’étant pas très connaisseur en assembleur en voulant ajouter de nombreux messages de debugg en copiant collant un write existant je ne modifiais pas la taille du message que je mettais ce qui causait ces erreurs d’affichages. Pour ce qui est de la segfault à la fin de l’exécution du programme elle était tout simplement du au fait qu’il n’y avait pas d’exit et donc le programme partait en segfault, la solution a bien évidemment été d’ajouter un exit au programme, et la solution pour les messages de debugg à été d’ajuster la taille dans les write en fonction des messages.

Avant changement de taille :



Le H affiché et le c sont tronqués et il y a une erreur de segmentation

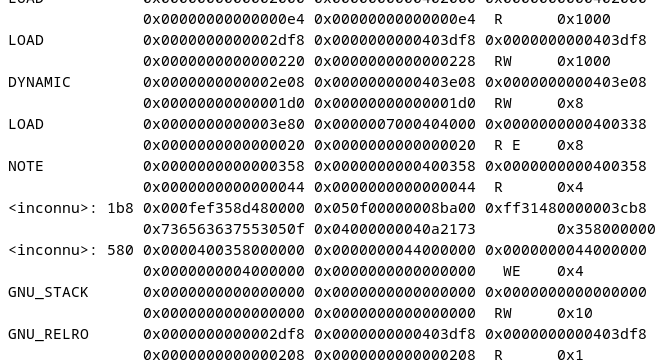
Après changement :



Les messages de debugg ne sont plus tronqués et plus de segfault.

Les problèmes beaucoup plus complexes ont commencé à la partie de l’injection dans le binaire. Le premier problème a été que mon ptload fraichement crée ne disposait pas des droits nécessaires pour executer le code malveillant, il a donc fallu modifier les flags dans le programm header du segment concerné avec la méthode discutée plus tôt.

Le deuxième problème était que lorsque je voulais injecter le shellcode dans le ptload je ne comprenais pas exactement comment était structuré un programm header et que la majorité des informations était importante ce qui ne laissait pas beaucoup de places pour notre shellcode et donc dépassait sur les autres segments ou réécrivait des parties importantes du header et cela corrompait le fichier.

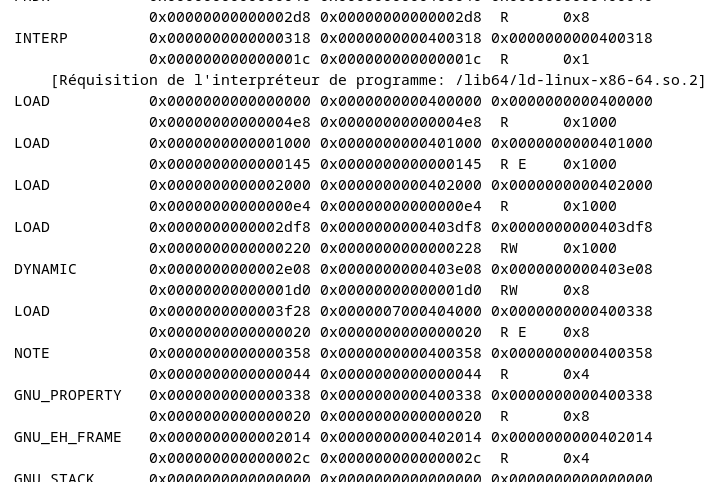


Cela à corrompu les segments suivantes en écrasant leurs données

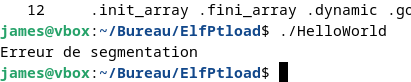
La solution que j’ai privilégié à été d’augmenter la taille du segment encore une fois dans le header ainsi que la taille de l’adressage mémoire memz et filesz. Cependant en augmentant cette taille et en injectant mon shellcode les segments d’après était écrasé par le shellcode ce qui rendait le fichier non valide.

Et j’ai aussi privilégié un shellcode directement en hexadécimal en pensant que cela serait plus facile à injecter à partir de la.

L’une des solutions auxquelles j’ai pensé a été de créer une boucle qui parcours les segments après notre segment cible et ajuste leurs offsets en fonction de la taille de notre injection pour que les segments ne soit pas écrasé mais décalé par l’écriture du shellcode et que le fichier et ces segments reste valident. Cependant cette solution n’a pas réellement été testé parce qu’elle me paraissait plus difficilement réalisable avec la façon dont je me déplace dans le fichier et que je n’ai pas un buffer qui contienne toutes les données du fichier. C’est pourquoi je suis parti sur une deuxième option qui consistait à changer l’offset non pas des segments d’après mais de celui du nouveau segment ptload et de le placer en toute fin de fichier pour que le shellcode puisse être écrit sans écraser quelque chose derrière lui. Pour cela j’ai modifié les segments de l’adressage virtuel pvaddr et la phoff qui contient l’offset du segment en lui donnant la valeur 16000 qui dans mon cas dépassait la taille du fichier même si c’est une information que l’on n’est pas censé avoir sauf si on mesure la taille du fichier mais cela n’a pas été implémenté donc cette technique était plus un test pour voir si l’injection pouvait marcher de cette manière et bien que cela n’écrase plus les segments d’après l’injection se résulte par un segfault à l’exécution du binaire infecté. Cela n’écrase plus les sections d’après.



Mais il y a une segfault :



Principales difficultés :

La principale difficulté que j’ai rencontrée est en réalité de comprendre comment fonctionnait l’assembleur notamment le fonctionnement des appels de fonctions avec les différents registres et de m’y familiariser et donc aussi passer du temps à apprendre l’utilisation des registres que ce soit dans ces appels de fonctions ou en dehors pour stocker des valeurs. La deuxième grande difficulté est la gestion de la mémoire qui provoque constamment des segmentation fault si on ne code pas rigoureusement. Dans mon cas cela concernait la compréhension du syscall read avec un buffer et qu’on ne pouvait pas accéder à des données déjà lues si elles ne sont plus dans un buffer ce qui était le cas puisque je réutilisais le même buffer pour analyser le segment suivant, donc en essayant d’accéder à ces adresses cela nous donnait automatiquement une segmentation fault. Et pour j’ai eu des difficultés sur l’injection et chercher à comprendre ou injecter le code et comment faire pour que cela soit au bon endroit et sans endommager le binaire pour qu’il reste exécutable.

Pour résumer les grandes étapes du code, le binaire ouvre un fichier ELF et lit les premiers octets et les comparent au magic number ELF pour savoir si c’es bien un fichier ELF ou non. Ensuite il récupère le nombre de programm headers et itère parmi ces programmes headers ou il compare à chaque fois leur type au type d’un segment PTNOTE. Si un segment PTNOTE est trouvé il change alors les droits d’exécution en read et exécutable et le type du segment en PTLOAD ainsi que la taille du segment et son offset a une adresse a la toute fin du fichier. Le Shell code est ensuite injecter à la fin du fichier.