Microcorruption Writeups

Hugo POLARD (Boycode)

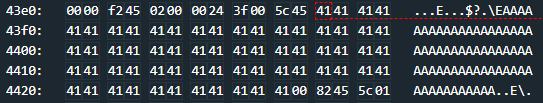
Notes :

* Le & avant une constante permet de sélectionner la valeur en mémoire à l’adresse de la constante directement.
  + &0x2400 => 2400 est une adresse
  + #0x2400 => 2400 est une valeure

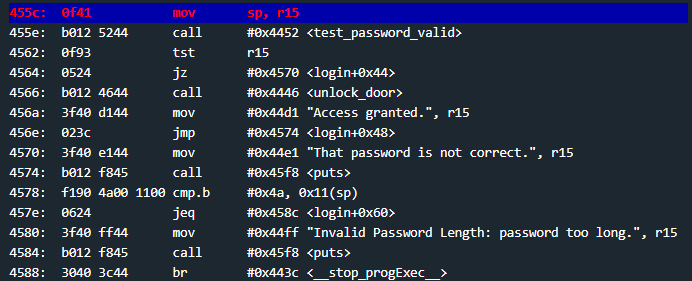
Johannesburg :

Dans ce challenge on utilise le module de sécurité HSM-1 qui prend en entrée le mot de passe et le vérifie puis retourne la validité ou non du mot de passe donné. Les mots de passe doivent faire entre 8 et 16 caractères mais on remarque que l’on peut aller en réalité jusqu’à 48 caractères.

Ces 48 caractères sont écrits en 0x2400 puis copiés sur la pile :



On peut imaginer overwrite l’adresse de retour de la fonction, mais cela est empêché par une comparaison qui envoie directement vers un stop\_progExec sans return :



On voit à l’adresse 0x4578 que la vérification est un comparaison du 17ème caractère de la chaine avec la constante 0x4a. On peut donc inclure dans notre payload cette valeur et passer la vérification directement pour avoir une exploitation classique de buffer overflow. Nous allons donc simplement rediriger le programme vers la fonction unlock\_door.

Pour cela il suffit simplement de mettre l’adresse de la fonction unlock\_door (en little endian) à la suite de notre octet permettant de bypass la vérification et du texte de dépassement de buffer.

Le payload final est donc : 41414141414141414141414141414141414a4644.

*Our operatives are entering the building.*

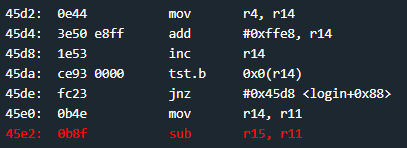
Santa Cruz :

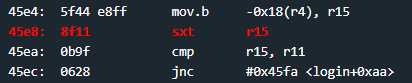
Nouveauté : Nécessité d’un username qui est aussi vérifié, amélioration de la vérification de la taille du mot de passe donné.

Tout d’abord, je regarde comment le programme vérifie la taille des inputs.

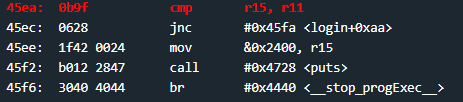
Après un getsn, les inputs sont copiés dans la pile par strcpy (comme précédemment). On retrouve alors dans la pile le username puis le password.

Je vois en premier une boucle qui compte le nombre de caractères du mot de passe et met sa taille dans r14 :

Ici r14 va pointer sur chaque caractère jusqu’au caractère nul et on lui enlève ensuite l’adresse du premier caractère stocké dans r15 pour avoir la taille.

On va ensuite mettre la valeur de l’octet se situant juste avant le mot de passe dans la pile dans r15 et le comparer à r11 (taille du mot de passe).

On peut donc en déduire que cette valeur précédant le mot de passe est sa taille maximale. En effet si la taille du mdp est plus grande que cette valeur on ne fait pas le jump et le programme se finit directement :



On a d’ailleurs la même mécanique ensuite pour vérifier la taille minimale qui est stockée sur la pile, dans l’octet précédant la taille maximale.

Comme la taille du username n’est pas vérifié et que celui-ci est stocké avant le mot de passe, mon objectif est donc d’overwrite la taille maximale du mot de passe avec l’username pour ensuite rediriger le programme vers la fonction unlock\_door.

Quelques adresses utiles :

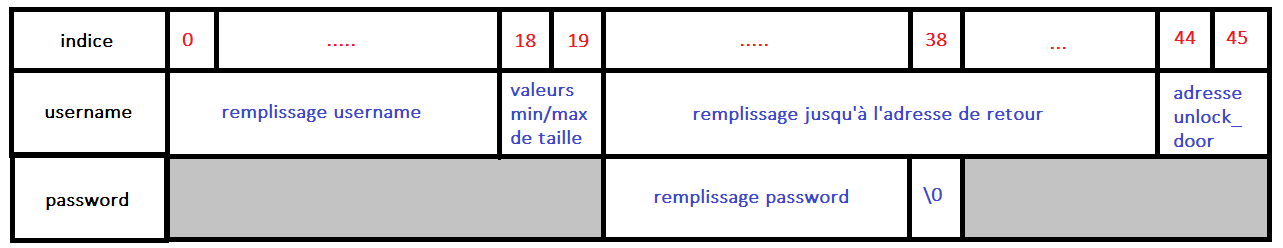
* Username : 0x43a2
* Taille minimale : 0x43b3 (attention à ne pas mettre une valeur trop grande)
* Taille maximale : 0x43b4
* Mot de passe : 0x43b5
* Octet qui doit être nul : 0x43c6 (explication plus bas)
* Adresse de retour de la fonction login : 0x43cc

On va donc remplir le username de 17 (0xb3-0xa2) caractères puis mettre la taille minimale désirée et enfin la taille maximale désirée.

Malheureusement, il y a une autre sécurité sur la taille du mot de passe : le programme vérifie que le 18ème caractère est bien un caractère nul. On ne peut donc pas directement réécrire l’adresse de retour car notre input sera coupé avant de l’atteindre (obligation d’avoir un 0x00). Il reste alors une possibilité : Réécrire cette adresse avec l’input **username** qui n’est pas vérifié en taille et aller ensuite mettre un caractère nul ou cela est nécessaire avec l’input **passsword**.

Il faudra alors faire attention à la taille min et max du mot de passe pour accéder à l’instruction « ret » de la fonction login et à bien mettre un caractère nul au 18ème caractère du mot de passe.

L’adresse de retour qu’on veut réécrire se trouve 23 octets après le mot de passe (0x43cc-0x43b5 = 0x17). Sa valeur est 0x4a44 (adresse de la fonction **unloock**\_**door**). On va donc avoir un payload de la forme :



Username : 505050505050505050505050505050505001EE51515151515151515151515151515151515151515151514a44

Password : 414141414141414141414141414141414100

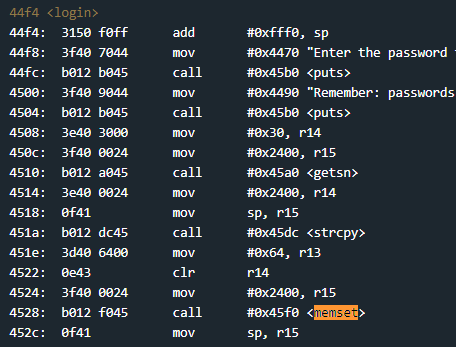
*Our operatives are entering the building.*

Montevideo :

Nouveauté : fonction memset

Je commence par me renseigner sur des problèmes liés à memset, et il y en a pas mal.

Unique appel à la fonction **memset** :

Définition de memset :  
Les registres r13, r14 et r15 servent certainement à transférer les paramètres de la fonction.

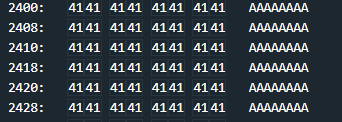
On a :  
r13 = #0x64  
r14 = #0x0  
r15 = #0x2400

Ce qui match avec le prototype de string.h :

void\* memset ( void \* , int , size\_t)

Contexte de memset : La fonction memset est appelée après l’entrée utilisateur et va écrire des 0 sur 64 bytes à partir de 0x2400. Note : on écrit précédemment à la même adresse.

Réception de l’entrée utilisateur : l’entrée utilisateur est mise dans la stack en 0x2400. 48 caractères sont stockés alors que le mot de passe fait entre 8 et 16 caractères…



Ces 48 caractères sont ensuite copiés dans un buffer situé à sp au moment de la copie sans limite de taille. Cette copie permet d’overwrite la stack des adresses **0x43ee** à 0x441d inclus. On réécrit donc les fonctions \_low\_level\_init, \_do\_copy\_data, et \_do\_clear\_bss.

En dépassant la taille du buffer, je remarque l’affichage d’un « adress unaligned », en effet on modifie alors directement le registre pc. On peut donc introduire un shell code et faire retourner l’exécution dessus. J’essaye alors de comprendre comment on a eu accès à pc.

En fait, le programme copie le buffer d’input dans la zone pointée par le stack pointer :



A la fin de la fonction login (fonction principale), le programme va sauter 16 octets pour passer la chaine et récupérer l’adresse de la fonction appelante, considérant que la chaine fait 16 caractères et non 48. Comme notre chaine d’input a était copié dans la stack juste avant, on va surement réécrire l’adresse de retour de la fonction login.



En faisant l’analyse dynamique du programme on confirme ce comportement :

En entrant dans login, l’adresse de sortie est bien mise sur la stack :

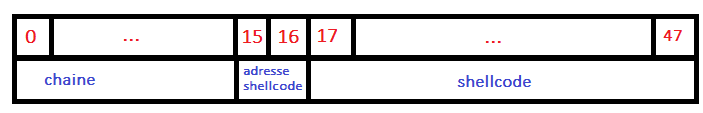


En s’arretant à l’appel de strcpy, on remarque que l’adresse pointée est bien 16 octets avant l’adresse de retour de la fonction login :



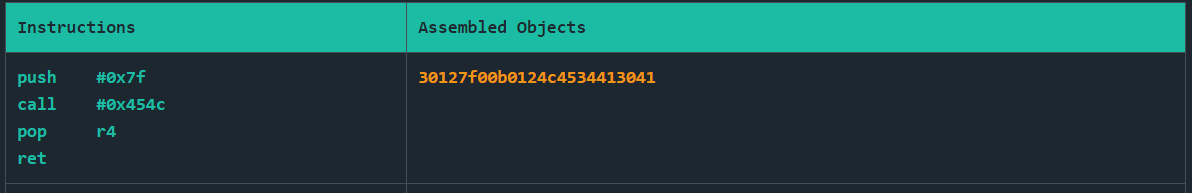
On passe donc à l’exploitation.

La chaine d‘exploitation aura la forme suivante :



AAAAAAAAAAAAAAAA 0044 < à générer >

On utilise l’assembleur donné par le site pour générer notre shellcode.



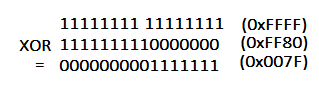
On a donc finalement le payload suivant (en hexadécimal ) :

41414141414141414141414141414141004430127f00b0124c45

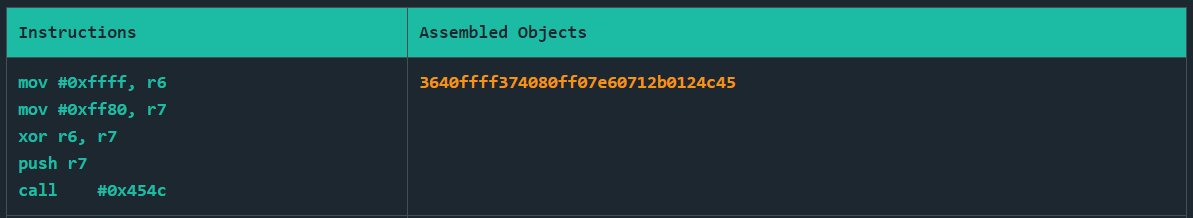
En testant, je remarque que seule la première partie est copiée, en effet je n’ai pas fais attention mais j’introduis une fin de chaine (0x00) dans la chaine, on va donc renvoyer vers 0x4402 et non 0X4400 en décalant le début de la shellcode dans le payload de deux octets (on ne doit agir par deux pour garder le « rythme » du processeur).

Le problème persiste, en effet il y a aussi un caractère nul dans le shellcode, ce qui est beaucoup plus problématique puisque qu’il vient de la valeur d’interruption, qui ne peut pas être modifiée.

Mon but est alors d’arriver à push la bonne valeur (0x7f) sans avoir de 0x00 dans mon payload. Pour cela je vais utiliser le XOR :



On utilise donc la shellcode suivante qui xor deux regitres et push le résultat sur la stack :



On a donc finalement le payload suivant :

41414141414141414141414141414141024441413640ffff374080ff07e60712b0124c45

*Our operatives are entering the building.*