Challenge : Pépin
Catégorie : Pwn

Énoncé :
Vous avez accès à une machine qui semble avoir un noyau Linux possédant un appel système #333 particulier.
Une fois connecté via SSH, utilisez le wrapper pour lancer le challenge.
Adresse : challenges1.france-cybersecurity-challenge.fr
Port : 4001, 4002 ou 4003 (au choix)
Utilisateur : ctf
Mot de passe : ctf

Table des matières

1) PREMIERE APPROCHE 2
2) ASSEMBLEUR 3
3) EXPLOIT 4

1) Première approche

En guise de première approche on peut commencer par observer les tags du challenge. On trouve ici « pwn » (logique) et « kernel ».
On peut ensuite se connecter via ssh et découvrir l'environnement du challenge.

```
SoEasY in ~/Bureau/FCSC_2020 [12:55]
> ssh ctf@challenges1.france-cybersecurity-challenge.fr -p 4001
ctf@challenges1.france-cybersecurity-challenge.fr's password:
ctf@pepin:~$ ls -al
total 3628
dr-xr-xr-x 1 ctf-admin ctf-admin    4096 Apr 25 10:46 .
drwxr-xr-x 1 ctf-admin ctf          4096 Apr 25 10:46 ..
-rw-r--r-- 1 ctf-admin ctf           220 May 15  2017 .bash_logout
-rw-r--r-- 1 ctf-admin ctf          3526 May 15  2017 .bashrc
-rw-r--r-- 1 ctf-admin ctf           675 May 15  2017 .profile
-rwxr-xr-x 1 ctf-admin ctf-admin    1248 Apr 25 10:45 .start.sh
-rw------- 1 ctf-admin ctf-admin 2645968 Apr 25 10:45 bzImage
-rw------- 1 ctf-admin ctf-admin 1026048 Apr 25 10:45 initramfs.cpio
-rwsr-x--- 1 ctf-admin ctf          8640 Apr 25 10:46 wrapper
-rw-r--r-- 1 ctf-admin ctf           126 Apr 25 10:45 wrapper.c
ctf@pepin:~$ cat wrapper.c
#include <unistd.h>
int main() {
  char *argv[] = {"/home/ctf/.start.sh", NULL};
  execve(argv[0], argv, NULL);
  return 0;
}
```

On retrouve donc ici divers fichiers dont le binaire « wrapper », compilé à partir du code source « wrapper.c ». On voit ici que le binaire fait appel à un script caché nommé « .start.sh ».

On peut ensuite lancer le wrapper.

```
ctf@pepin:~$ ./wrapper
------------------------------------------------------------------------------
To ease your exploit development, a secret folder shared between the host and
the vm will be created. You can access it at /mnt/share within the vm, and at
/tmp/tmp.hYLaLvQFb8 in the host. The folder will be deleted afterwards.
------------------------------------------------------------------------------

Press <Enter> to continue...

FCSC KERNEL

/ $ ls -al
total 4
drwxr-xr-x   13 root     root             0 May  2 10:58 .
drwxr-xr-x   13 root     root             0 May  2 10:58 ..
drwxr-xr-x    2 root     root             0 Feb 28 12:30 bin
drwxr-xr-x    3 root     root             0 May  2 10:58 dev
drwxr-xr-x    2 root     root             0 May  2 10:58 etc
drwxr-xr-x    3 root     root             0 Feb 28 12:30 home
----------    1 root     root          1762 Apr  6 16:04 init
drwxr-xr-x    3 root     root             0 May  2 10:58 mnt
dr-xr-xr-x   28 root     root             0 May  2 10:58 proc
drwx------    2 root     root             0 Apr  6 14:01 root
drwxr-xr-x    2 root     root             0 May  2 10:58 run
dr-xr-xr-x   10 root     root             0 May  2 10:58 sys
drwxr-xr-x    2 root     root             0 May  2 10:58 tmp
drwxr-xr-x    3 root     root             0 May  2 10:58 var
/ $
```

On se retrouve ainsi dans un nouvel environnement et on nous annonce la présence d'un dossier partagé temporaire créé dans /mtn/share dans ce nouvel environnement et dans /tmp/.<chaine aléatoire> sur la machine de base.

Allons voir du côté du script pour mieux comprendre ce qu'il se passe.

```
/ $ exit
reboot: Power down
ctf@pepin:~$ cat .start.sh
#!/bin/bash -p

STTY=$(stty -g)
stty intr ^-

TEMP=$(mktemp -d)
chgrp ctf ${TEMP}
chmod 730 ${TEMP}

echo "------------------------------------------------------------------------------"
echo "To ease your exploit development, a secret folder shared between the host and"
echo "the vm will be created. You can access it at /mnt/share within the vm, and at"
echo "${TEMP} in the host. The folder will be deleted afterwards."
echo "------------------------------------------------------------------------------"
echo ""
read -p "Press <Enter> to continue..."

timeout --foreground 300 qemu-system-x86_64                    \
    -m 64M                                                     \
    -cpu kvm64                                                 \
    -kernel bzImage                                            \
    -nographic                                                 \
    -append 'console=ttyS0 loglevel=3 oops=panic panic=1 kaslr' \
    -initrd initramfs.cpio                                     \
    -monitor /dev/null                                         \
    -fsdev local,id=exp1,path=${TEMP},security_model=mapped    \
    -device virtio-9p-pci,fsdev=exp1,mount_tag=ecsc

rm -rf "${TEMP}" 2> /dev/null
stty "${STTY}"
```

On voit donc que le script émule une distribution linux avec un kernel custom nommé « bzImage » avec un dossier partagé créé avec « mktemp -d ».

2) Assembleur

L'énoncé du challenge nous incite à exécuter l'appel système #333.
On va pour cela écrire un petit programme en assembleur qui va effectuer cet appel système (ou « syscall »).

Ici mon programme va afficher un message qui n'a pas d'importance en guise de contrôle de la bonne exécution du programme, puis va effectuer le fameux appel système #333 avant de quitter.

Pour effectuer cet appel système, il suffit de mettre « 333 » dans rax avant le « syscall ».
(Pour ce qui est du message, on met 1 dans rax car c'est le premier appel système, 1 dans rdi pour « stdout », le message dans rsi et la taille du message dans rdx.)

```
1   BITS 64
2
3   section .data
4       hello db `Syscall 333 incoming...`
5       hello_len equ $-hello
6
7   section .text
8       global _start
9
10  _start:
11      mov rax, 1
12      mov rdi, 1
13      mov rsi, hello
14      mov rdx, hello_len
15      syscall
16
17  _leSyscall:
18      mov rax, 333
19      syscall
20
21  _exit:
22      mov rax, 60
23      mov rdi, 0
24      syscall
```

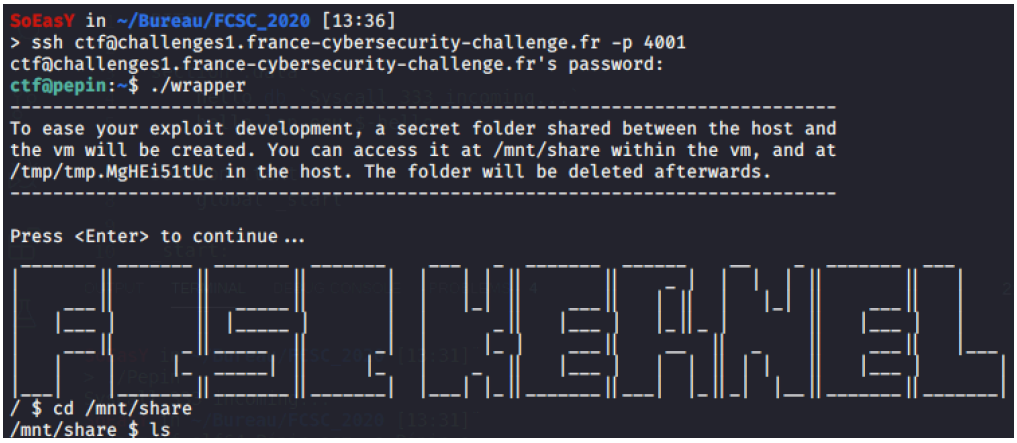
On peut ensuite assembler ce programme et le linker pour produire un ELF 64 bits exécutable sur la distribution Linux émulée avec noyau modifié.
On le test ensuite en local pour vérifier son bon fonctionnement.

```
SoEasY in ~/Bureau/FCSC_2020 [13:31]
> nasm -f elf64 Pépin.asm -o Pépin.o

SoEasY in ~/Bureau/FCSC_2020 [13:31]
> ld Pépin.o -o Pepin

SoEasY in ~/Bureau/FCSC_2020 [13:31]
> ./Pepin
Syscall 333 incoming...
```

3) Exploit

Pour exécuter notre programme sur la distribution émulée, on va lancer le wrapper, noter le dossier temporaire créer et envoyer notre fichier dans ce même dossier afin de le retrouver dans /mnt/share.

Pour ceci, on peut utiliser « scp » (Secure CoPy) car on a les identifiants ssh.



On note ici que le dossier temporaire est « /tmp/.MgHEi51tUc ». On va donc envoyer notre programme dans celui-ci.

```
SoEasY in ~ [13:36]
> scp -P 4001 -r -p /root/Bureau/FCSC_2020/Pepin ctf@challenges1.france-cybersecurity-challenge.fr:/tmp/tmp.MgHEi51tUc
ctf@challenges1.france-cybersecurity-challenge.fr's password:
Pepin                                                    100% 9000    62.0KB/s   00:00
```

On peut donc ensuite récupérer notre programme dans /mnt/share et l'exécuter.

```
/mnt/share $ ls
Pepin
/mnt/share $ ./Pepin
Syscall 333 incoming ... /mnt/share $
```

Ensuite commence le début d'une longue exploration car, à première vue, rien n'a changé.

Un « ls -alR | grep flag » et autre commandes naïves plus tard, on peut ensuite penser à se tourner du côté des commandes liées au kernel.

Enfin, en affichant la mémoire tampon du message de kernel avec la commande « dmesg » le flag nous apparait.

```
Freeing unused kernel memory: 2016K
Freeing unused kernel memory: 784K
tsc: Refined TSC clocksource calibration: 3311.977 MHz
clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x2fbd7e467c1, max_idle_ns: 440795237359 ns
clocksource: Switched to clocksource tsc
input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3
random: fast init done
FCSC{b820fd6ce2365286396c923b899477577b0b97036a37ace0e93fd6b628d833ad}
```

Challenge terminé ! Un challenge relativement facile mais très divertissant.