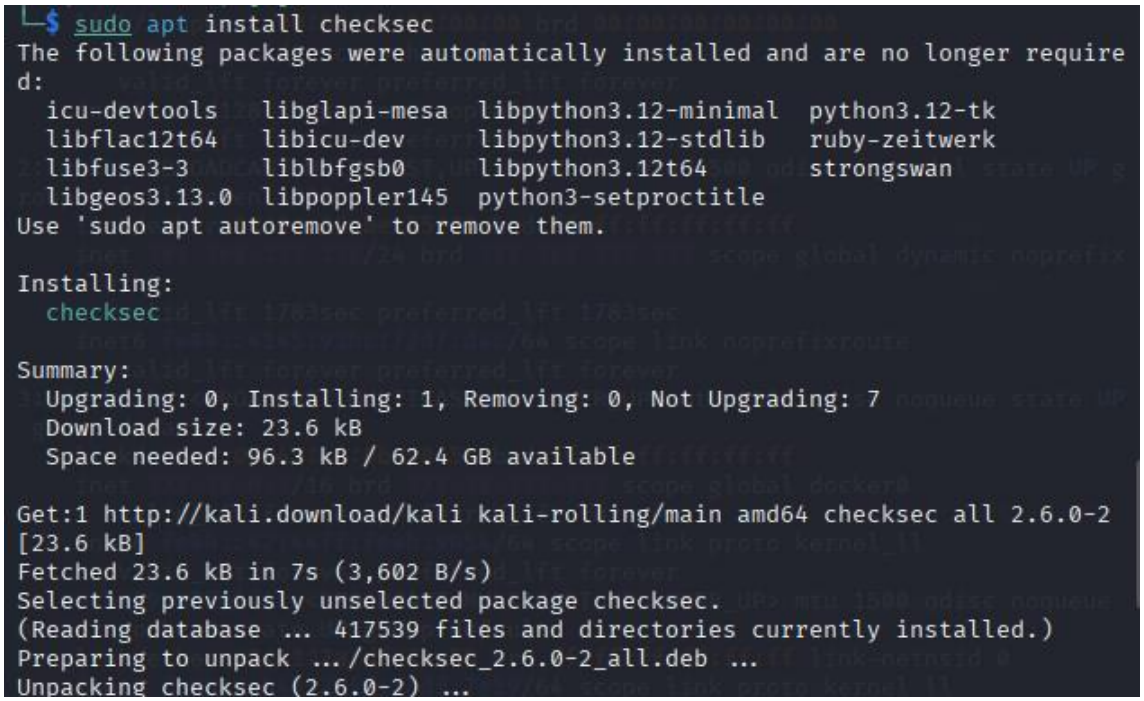
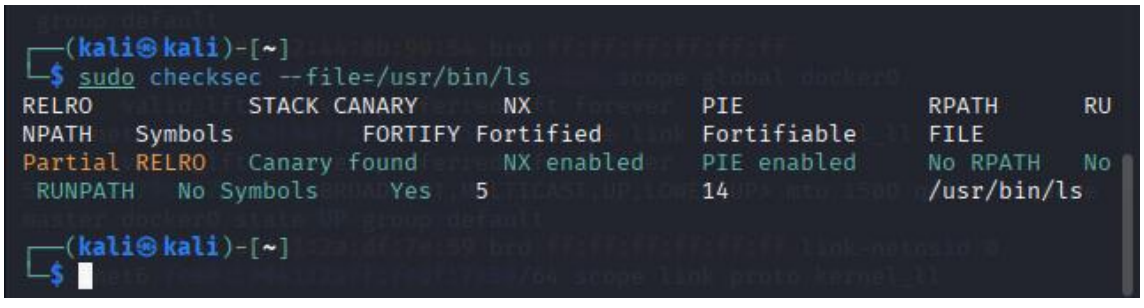
CHECKSEC

First of all I install checksec



Then we analize the comando ls:



Now we créate the file hello.c, then we compile it:

```
┌──(kali㉿kali)-[~]
└─$ sudo nano hello.c

┌──(kali㉿kali)-[~]
└─$ ls
2.7  connection_component.py  Desktop  Documents  Downloads  hello.c  leerWebSocket.py  Music  Pictures  Public  __pycache__  Templates  venv  Videos  WebSocketServer.py

┌──(kali㉿kali)-[~]
└─$ sudo gcc hello.c -o hello

┌──(kali㉿kali)-[~]
└─$ ll
total 76
-rw-rw-r-- 1 kali kali  1219 Apr 22 11:54 2.7
-rw-rw-r-- 1 kali kali  1027 Apr 23 12:19 connection_component.py
drwxr-xr-x 3 kali kali  4096 Apr 23 11:50 Desktop
drwxr-xr-x 2 kali kali  4096 Apr 22 11:03 Documents
drwxr-xr-x 2 kali kali  4096 Apr 22 11:03 Downloads
-rwxr-xr-x 1 root root 15952 May 19 09:58 hello
-rw-r--r-- 1 root root    71 May 19 09:58 hello.c
-rw-rw-r-- 1 kali kali   517 Apr 23 12:20 leerWebSocket.py
drwxr-xr-x 2 kali kali  4096 Apr 22 11:03 Music
drwxr-xr-x 2 kali kali  4096 Apr 22 11:03 Pictures
drwxr-xr-x 2 kali kali  4096 Apr 22 11:03 Public
drwxrwxr-x 2 kali kali  4096 Apr 23 12:19 __pycache__
drwxr-xr-x 2 kali kali  4096 Apr 22 11:03 Templates
drwxrwxr-x 5 kali kali  4096 Apr 23 11:54 venv
drwxr-xr-x 2 kali kali  4096 Apr 22 11:03 Videos
-rw-rw-r-- 1 kali kali  2147 Apr 23 12:23 WebSocketServer.py
```

We analize it with file and checksec

```
┌──(kali㉿kali)-[~]
└─$ file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=59ede49dcb30147f96252b6f2384a209c1e55162, for GNU/Linux 3.2.0, not stripped

┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   36 Symbols      No    0                0               ./hello
```

The next step is installing jq and try these checksec functions:

```
┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello --output=json | jq | grep symbols
    "symbols": "yes",

┌──(kali㉿kali)-[~]
└─$ sudo checksec --file=/usr/bin/ls --output=json | jq
{
  "/usr/bin/ls": {
    "relro": "partial",
    "canary": "yes",
    "nx": "yes",
    "pie": "yes",
    "rpath": "no",
    "runpath": "no",
    "symbols": "no",
    "fortify_source": "yes",
    "fortified": "5",
    "fortify-able": "14"
  }
}
```

To avoid security problems and data loses, we run these command:

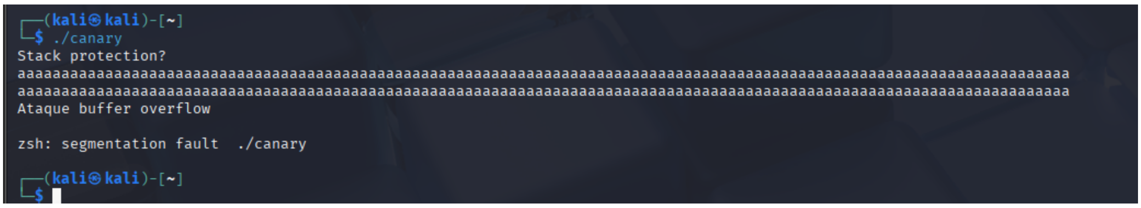
```
┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello --output=json | jq | grep symbols
    "symbols": "yes",

┌──(kali㉿kali)-[~]
└─$ gcc -s hello.c -o hello

┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello --output=json | jq | grep symbols
    "symbols": "no",
```

To try stack and buffer overflow we créate de file canary.c, but in the momento of compilation it gives us this issue:

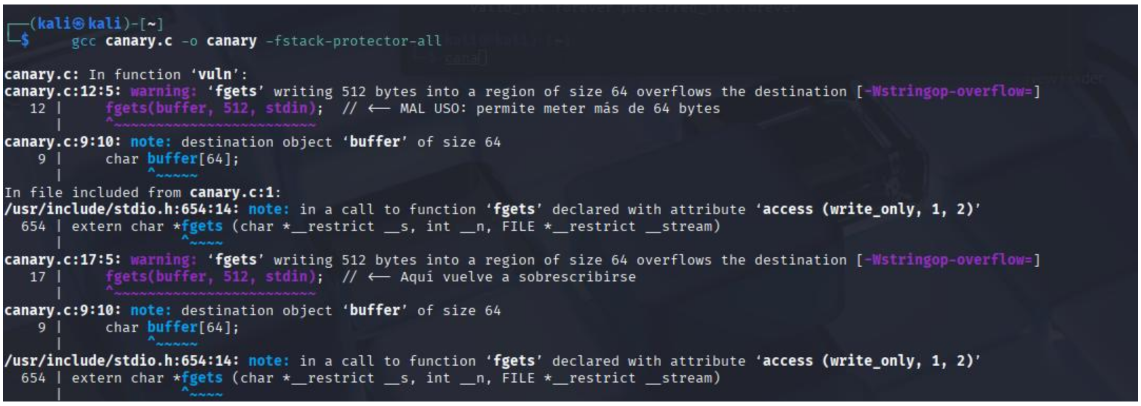
```
┌──(kali㉿kali)-[~]
└─$ sudo gcc canary.c -o canary
canary.c: In function ‘vuln’:
canary.c:9:5: error: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
    9 |     gets (bufer);
      |     ^~~~
      |     fgets
```

I asked ChatGpt to remake the code not using the function gets:

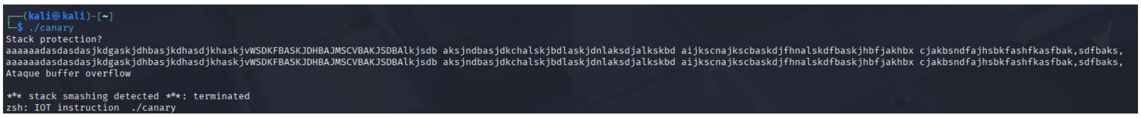
```c
#include <stdio.h>
#include <string.h>

void hacked() {
    puts("¡Función que no llama nadie!");
}

void vuln() {
    char buffer[64];

    puts("Stack protection?");
    fgets(buffer, 512, stdin);  // <-- MAL USO: permite meter más de 64 bytes

    printf("%s", buffer);
    puts("Ataque buffer overflow");

    fgets(buffer, 512, stdin);  // <-- Aquí vuelve a sobrescribirse
}

int main() {
    vuln();
    return 0;
}
```



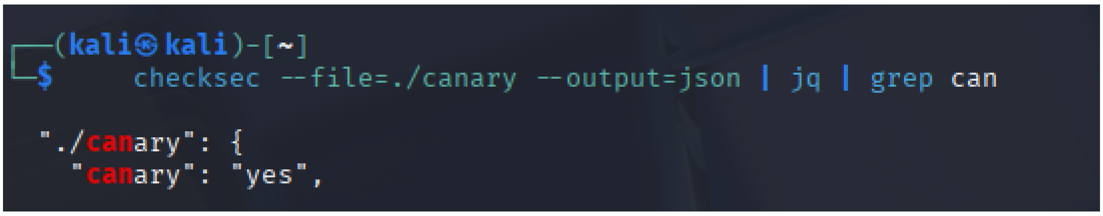
To protect the program when compiling, we add this code in the compilation process:



Now, when we run the program, it appears this:



And if we prove it with checksec:



Now we run the following commands, these commands use checksec to verify whether the binary has PIE (Position Independent Executable) enabled, and show that compiling with -no-pie disables it.

```
┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello --output=json | jq | grep pie
    "pie": "yes",

┌──(kali㉿kali)-[~]
└─$ checksec --file=/bin/ls --output=json | jq | grep pie
    "pie": "yes",

┌──(kali㉿kali)-[~]
└─$ gcc -no-pie hello.c -o hello

┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello --output=json | jq | grep pie
    "pie": "no",
```

Now we run this comando that checks if the binary hello has NX (Non-eXecutable stack) protection enabled, and confirms it with "nx": "yes".

```
┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello --output=json | jq | grep nx
    "nx": "yes",
```

We run this commands that when compiling with -Wl,-z,relro,-z,now enables Full RELRO (Relocation Read-Only), improving binary security compared to the default Partial RELR

```
┌──(kali㉿kali)-[~]
└─$ gcc hello.c -o hello

┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    PIE enabled   No RPATH   No RUNPATH   36 Symbols      No    0                0               ./hello

┌──(kali㉿kali)-[~]
└─$ gcc -Wl,-z,relro,-z,now hello.c -o hello

┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
Full RELRO      No canary found   NX enabled    PIE enabled   No RPATH   No RUNPATH   36 Symbols      No    0                0               ./hello
```

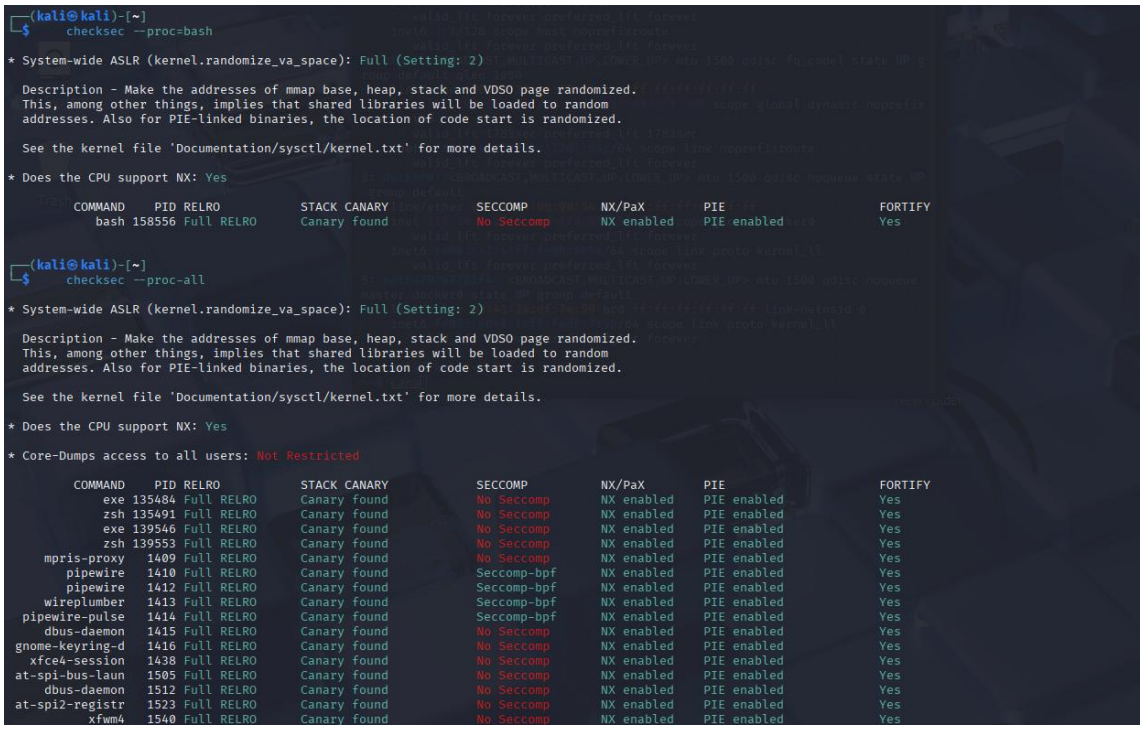
Now this comand compiles with _FORTIFY_SOURCE=2 and optimization, but since no fortifiable functions are used, checksec shows no FORTIFY protection applied.

```
┌──(kali㉿kali)-[~]
└─$ gcc -D_FORTIFY_SOURCE=2 -O2 hello.c -o hello

┌──(kali㉿kali)-[~]
└─$ checksec --file=./hello
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    PIE enabled   No RPATH   No RUNPATH   36 Symbols      No    0                0               ./hello
```

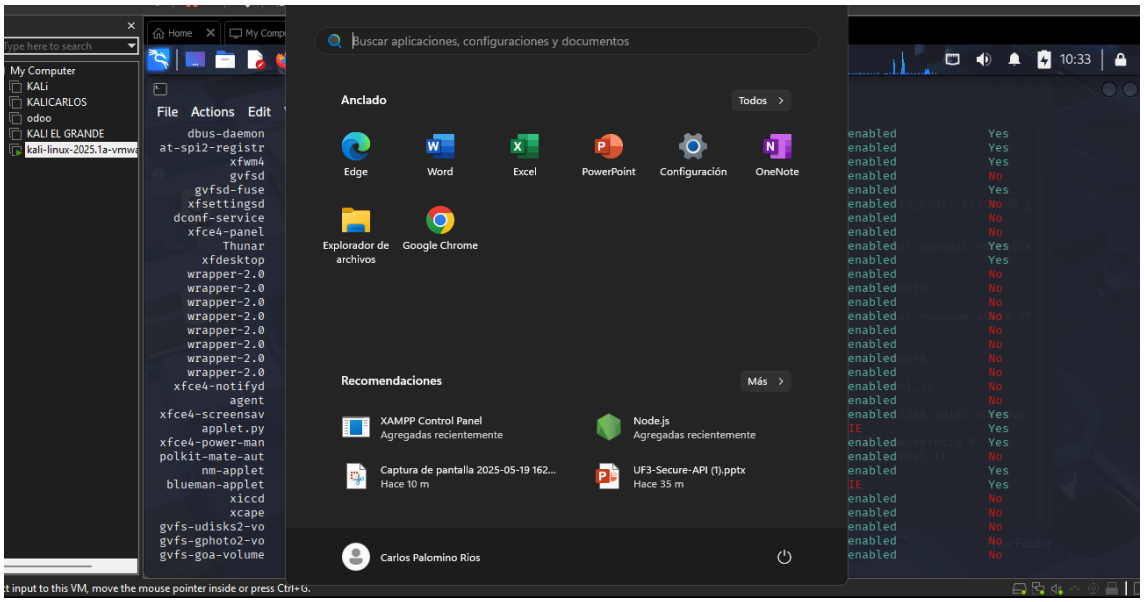
We upload the canary.c file using fgets insteado of gets and deleteing the line jump.

```c
#include <stdio.h>
#include <string.h>
void hacked () {    puts ("Funcion que no llama nadie"); }

void vuln () {
    char bufer[64];
    puts ("stack protection");
    // Reemplazar gets con fgets
    fgets (bufer, sizeof(bufer), stdin);
    // Eliminar el salto de linea del final de bufer
    bufer[strcspn(bufer, "\n")] = '\0';
    printf("\n%s" , bufer);
    puts("\nAtaque buffer overflow");
}
int main(){    vuln(); }
```

Finally we run the following comands with checksec to see all thats its executing:



```
┌──(kali㉿kali)-[~]
└─$ checksec --proc=bash

* System-wide ASLR (kernel.randomize_va_space): Full (Setting: 2)

  Description - Make the addresses of mmap base, heap, stack and VDSO page randomized.
  This, among other things, implies that shared libraries will be loaded to random
  addresses. Also for PIE-linked binaries, the location of code start is randomized.

  See the kernel file 'Documentation/sysctl/kernel.txt' for more details.

* Does the CPU support NX: Yes

        COMMAND    PID RELRO           STACK CANARY      SECCOMP         NX/PaX      PIE             FORTIFY
           bash 158556 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes


┌──(kali㉿kali)-[~]
└─$ checksec --proc-all

* System-wide ASLR (kernel.randomize_va_space): Full (Setting: 2)

  Description - Make the addresses of mmap base, heap, stack and VDSO page randomized.
  This, among other things, implies that shared libraries will be loaded to random
  addresses. Also for PIE-linked binaries, the location of code start is randomized.

  See the kernel file 'Documentation/sysctl/kernel.txt' for more details.

* Does the CPU support NX: Yes

* Core-Dumps access to all users: Not Restricted

        COMMAND    PID RELRO           STACK CANARY      SECCOMP         NX/PaX      PIE             FORTIFY
            exe 135484 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
            zsh 135491 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
            exe 139546 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
            zsh 139553 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
    mpris-proxy   1409 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
       pipewire   1410 Full RELRO      Canary found      Seccomp-bpf     NX enabled  PIE enabled     Yes
       pipewire   1412 Full RELRO      Canary found      Seccomp-bpf     NX enabled  PIE enabled     Yes
    wireplumber   1413 Full RELRO      Canary found      Seccomp-bpf     NX enabled  PIE enabled     Yes
  pipewire-pulse  1414 Full RELRO      Canary found      Seccomp-bpf     NX enabled  PIE enabled     Yes
     dbus-daemon  1415 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
 gnome-keyring-d  1416 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
   xfce4-session  1438 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
at-spi-bus-laun  1505 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
     dbus-daemon  1512 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
at-spi2-registr  1523 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
          xfwm4  1540 Full RELRO      Canary found      No Seccomp      NX enabled  PIE enabled     Yes
```



To prove that this is my pc, because the virtual machine is literally a Kali prebuilt VM downloaded from the oficial website, in this screenshot I show my name Carlos Palomino Rios with the VM behind running the comands of the practice.