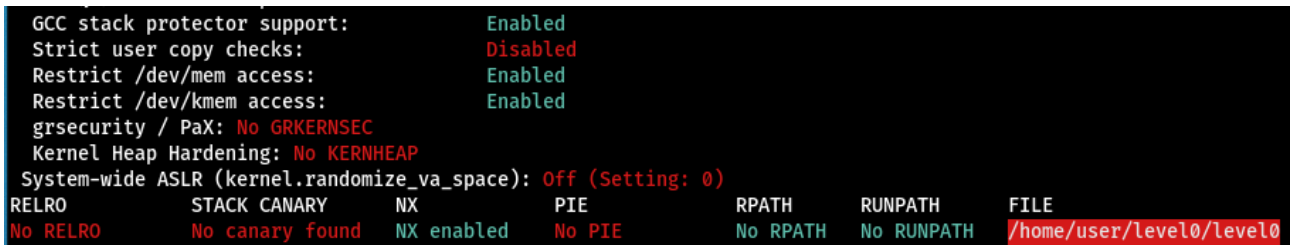
42 PROJECT
Rainfall

0. Level 0

On trouve, comme prévu, dans le dossier **home** de cet utilisateur, le binaire à exploiter. Une commande listant les protections en place pour ce binaire avec **checksec** est lancée à la connexion SSH et nous donne le résultat suivant :



- On a NX activé, donc pas de shellcode sur la stack.
- Pas de PIE, donc les instructions de la section .text vont se trouver à des adresses virtuelles fixes.
- Pas d'ASLR sur le système, les potentielles libairires partagées se situeront donc également à des adresses fixes.

Lorsqu'on tente de le lancer sans argument, le programme nous renvoie une segfault ; lorsqu'on tente de le lancer avec un argument aléatoire "aaaa", un simple message "`No!`" nous est renvoyé.

On transfert le binaire sur notre machine Kali pour l'examiner de plus près en utilisant netcat. On utilise **edb** pour débugger (gdb est un outil très versatile mais l'interface graphique de **edb** rend la compréhension tout de même plus simple. En plus edb est en syntaxe Intel par défaut) :

```
edb --run level0 4242
```

On a un binaire 32 bits, donc rappelons-nous qu'en assembleur, les arguments des fonctions sont passés sur la stack.

En examinant le code en assembleur, on aperçoit de manière générale :

- Un call à `atoi`. L'argument sur la stack juste avant l'appel nous informe que l'argument est `argv[1]`.
- Une comparaison du résultat de atoi stocké dans $eax :
  - Si `$eax` est égal à `0x1a7`, alors un appel à `getgid`, `getuid`, `setresgid`, `setresuid` est effectué (pas hyper important).
    De plus, un appel à `execv` est effectué, avec en argument "`/bin/sh`" et NULL.
  - Si `$eax` n'est pas égal à `0x1a7`, alors un appel à `fwrite` est effectué `fwrite("No !\n", 1, 5, stdout)`.

Ce qui nous donne le code source reconstruit suivant :

```c
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int     main(int argc, char **argv)
{
    int a;
    a = atoi(argv[1]);

    if (a == 0x1a7)
    {
        char          *s = strdup("/bin/sh");
        gid_t         gid = getegid();
        uid_t         uid = geteuid();

        setresgid(gid, gid, gid);
        setresuid(uid, uid, uid);
        execv("/bin/sh", 0);
    }
    else
        fwrite("No !\n", 1, 5, stdout);
    return (0);
}
```

D'ici l'exploit est très simple, il suffit de passer au programme, en premier argument, une valeur qui, traduite par atoi, sera égale à 0x1a7 (= 423 en décimal) :

```
./level0 423
```

On récupère une shell et on peut passer **au niveau suivant**.

>> Exploitation manuelle

```
./level0 423
```

>> Exploit automatique

```python
from pwn import *

s = ssh(host='192.168.1.45', port=4242, user="level0", password="level0")
p = s.process(['/home/user/level0/./level0', '423'])

p.interactive()
```

Got flag.