# Sécurité Applicative - Laboratoire 1 (4h)

Adrien Voisin, Bastien Bodart

IR313 - Henallux 2021-2022

## 1 Permissions spéciales sous Linux

La plupart des distributions *Linux* offrent la possibilité de configurer des permissions spéciales sur des fichiers exécutables. Cela apporte une certaine flexibilité aux utilisateurs mais peut également ouvrir la porte à l'exploitation de logiciels vulnérables.

1. Ajoutez un utilisateur sur votre machine

Adduser vs useradd :

- adduser --> interactive
- useradd --> on donne nous-même l'info


2. Aidez-vous du fichier */etc/sudoers* afin de donner le droit à l'utilisateur d'exécuter le programme de l'exercice 12 du laboratoire 0 en *root*, et sans devoir spécifier de mot de passe.

https://github.com/carrot827/S-curit-applicative-IR-B3-Q1-2021-2022/blob/labo0/ex12.c

Sans rien modifier :

```
┌──(toto㉿host)-[/home/user/Documents]
└─$ sudo ./hello filetoreadtest

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for toto:
toto is not in the sudoers file.  This incident will be reported.

┌──(toto㉿host)-[/home/user/Documents]
└─$
```

On modifie :

```
  GNU nano 5.4                                  /etc/sudoers
#
# This file MUST be edited with the 'visudo' command as root.
#
# Please consider adding local content in /etc/sudoers.d/ instead of
# directly modifying this file.
#
# See the man page for details on how to write a sudoers file.
#
Defaults        env_reset
Defaults        mail_badpass
Defaults        secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL
toto    ALL=NOPASSWD: /home/user/Documents/hello
# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL
```

Et maintenant, on sait l'exécuter avec toto, sans mot de passe.

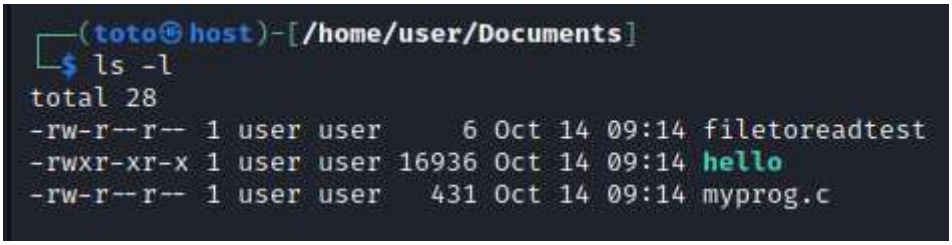
3. Utilisez les droits *SUID* [1] pour permettre à l'utilisateur de lancer le programme de l'exercice 12 du laboratoire 0 avec les mêmes droits que le propriétaire du binaire. Pour cela, faites au préalable une copie du binaire depuis l'utilisateur root.

On se met en root :

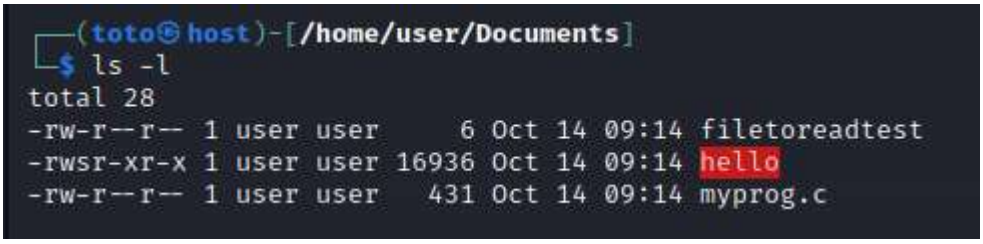
```
┌──(root㉿host)-[/home/user/Documents]
└─# chmod u+s hello
```

Puis on retourne sur Toto

Avant modification :




Après modification :




# 2 Introduction à gdb

1. Sélectionnez 2 exercices développés pour la séance de laboratoire 0 et compilez-les avec l'option $-g$ de *gcc* (cela compile le fichier binaire en mode debug, ce qui vous permettra d'utiliser toutes les fonctionnalités de *gdb*). Pour chaque exécutable, explorez les fonctions suivantes de *gdb*:

   - Affichez le code source à l'aide de la commande *list*

**Compiler**


*-g est le mode debug*

**Installer gdb :**
sudo apt-get install gdb

```
┌──(user㉿host)-[~/Documents]
└─$ gdb hello
GNU gdb (Debian 10.1-2) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...
(gdb)
```

**list**

```
(gdb) list
10              if(file==NULL)
11              {
12                      printf("Erreur");
13                      exit(EXIT_FAILURE);
14              }
15              while(fgets(ligne, MAXLINE, file) != NULL)
16              {
17                      fputs(ligne,stdout);
18              }
19              fclose(file);
(gdb)
20      }
21
22      void main(int argc, char *argv[])
23      {
24              if(argc != 2)
25              {
26                      printf("Erreur");
27                      exit(EXIT_FAILURE);
28              }
29
(gdb)
30              aouvrir(argv[1]);
31      }
(gdb)
```

**list main**

```
(gdb) list main
18              }
19              fclose(file);
20      }
21
22      void main(int argc, char *argv[])
23      {
24              if(argc != 2)
25              {
26                      printf("Erreur");
27                      exit(EXIT_FAILURE);
(gdb)
28              }
29
30              aouvrir(argv[1]);
31      }
(gdb)
Line number 32 out of range; myprog.c has 31 lines.
(gdb)
```

- Ajoutez des points d'arrêts à l'aide de la commande *break*

```
(gdb) break main
Breakpoint 1 at 0x1218: file myprog.c, line 24.
(gdb)
```

- Parcourez le code à l'aide des commandes *next*, *step*, et *continue*. Quelle est la différence entre les commandes *next* et *step*?

On tape : « layout prev ».
Ensuite on peut taper les commandes « next » et « step ».

**next**

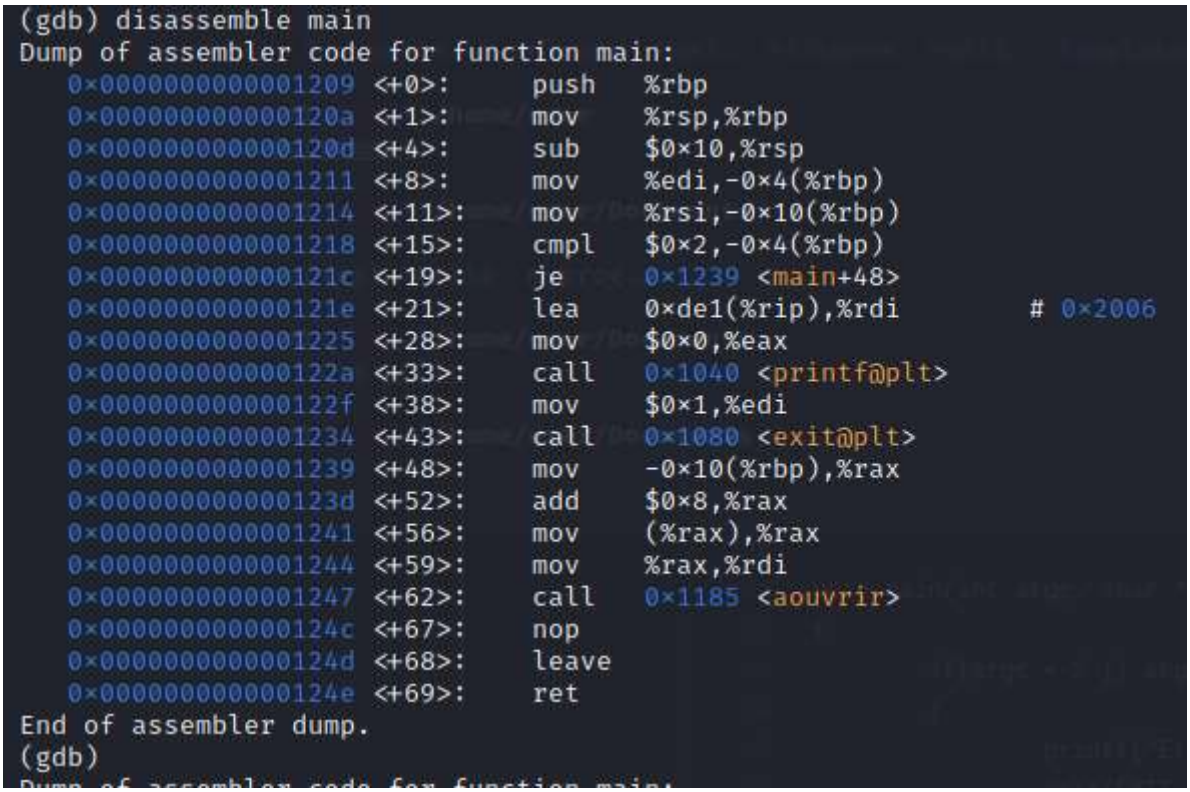
Next avance ligne de code par ligne de code en mode debug

Sans le mode debug, on avance instruction par instruction.

**step**

step avance par instruction processeur.

- Affichez le code assembleur de vos programme à l'aide de la commande *disassemble*



```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001209 <+0>:     push   %rbp
   0x000000000000120a <+1>:     mov    %rsp,%rbp
   0x000000000000120d <+4>:     sub    $0x10,%rsp
   0x0000000000001211 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000000000001214 <+11>:    mov    %rsi,-0x10(%rbp)
   0x0000000000001218 <+15>:    cmpl   $0x2,-0x4(%rbp)
   0x000000000000121c <+19>:    je     0x1239 <main+48>
   0x000000000000121e <+21>:    lea    0xde1(%rip),%rdi        # 0x2006
   0x0000000000001225 <+28>:    mov    $0x0,%eax
   0x000000000000122a <+33>:    call   0x1040 <printf@plt>
   0x000000000000122f <+38>:    mov    $0x1,%edi
   0x0000000000001234 <+43>:    call   0x1080 <exit@plt>
   0x0000000000001239 <+48>:    mov    -0x10(%rbp),%rax
   0x000000000000123d <+52>:    add    $0x8,%rax
   0x0000000000001241 <+56>:    mov    (%rax),%rax
   0x0000000000001244 <+59>:    mov    %rax,%rdi
   0x0000000000001247 <+62>:    call   0x1185 <aouvrir>
   0x000000000000124c <+67>:    nop
   0x000000000000124d <+68>:    leave
   0x000000000000124e <+69>:    ret
End of assembler dump.
(gdb)
```

...

...

# 3 Analyse de la stack avec gdb

Maintenant que vous maitrisez l'outil *gdb*, nous allons analyser le contenu de la mémoire durant l'exécution d'un programme. Pour cet exercice nous utiliserons le programme définit ci-dessous:

```c
#include <stdio.h>

int add(int, int);
int mul(int, int);

int main()
{
    int four, three, result; // breakpoint 1

    four = 4;
    three = 3;

    result = mul(four, three);

    printf("%d x %d = %d\n", four, three, result);

    return 0; // breakpoint 4
}


int add(int a, int b)
{
    int first, second, result; // breakpoint 3

    first = a;
    second = b;

    result = first + second;

    return result;
}


int mul(int x, int y)
{
    int number, multiplier, result, i; // breakpoint 2

    number = x;
    multiplier = y;

    result = 0;
    i = 0;

    while (i < multiplier)
    {
        result = add(result, number);
        i = i + 1;
    }

    return result;
}
```

| Breakpoint | rsp | rbp |
|---|---|---|
| breakpoint 1 | 0×7fffffffdfa0 | 0×7fffffffdfb0 |
| Breakpoint 2 | 0×7fffffffdf78 | 0×7fffffffdf90 |
| Breakpoint 3 | 0×7fffffffdf68 | 0×7fffffffdf68 |
| Breakpoint 3 | 0×7fffffffdf68 | 0×7fffffffdf68 |
| Breakpoint 3 | ... | ... |
| Breakpoint 4 | ... | ... |

1. Compilez ce programme avec l'argument -g et désactivez la gestion aléatoire des adresses en mémoire :

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

**A taper en root**

2. Avec *gdb*, vous allez placer un point d'arrêt comme indiqué ci-dessus dans le code. Pour chaque point d'arrêt, indiquez l'adresse contenant *rbp* et *rsp* (A l'aide de la commande *print*). Utilisez le tableau afin d'enregistrer ces adresses. A partir de là, représentez sur papier l'évolution de la *stack* au cours de l'exécution.

gdb ex2

list

Ensuite, il faut des breakpoint en fonction de l'objectif vise, ici aux lignes ou c'est demandé dans le code donné plus haut.

Ici, le premier est à la ligne 8. Pour le lancer, on fait « run »

```
(gdb) break 8
Breakpoint 1 at 0×113d: file codepage2.c, line 10.
(gdb) run
Starting program: /home/user/Documents/ex2

Breakpoint 1, main () at codepage2.c:10
10              four = 4;
(gdb)
```

Ils demandent des breakpoint aux lignes :

8, 17, 22 et 34

On les ajoute avec : « break 8 », « break 17 », …

Ensuite :

run

print $rsp

print $rbp

continue

print $rsp

print $rbp
continue
etc …

*Information : L'instruction « next » avance ligne par ligne*

1

```
(gdb) print $rsp
$1 = (void *) 0x7fffffffdfa0
(gdb) print $rbp
$2 = (void *) 0x7fffffffdfb0
(gdb)
```

2

```
(gdb) continue
Continuing.

Breakpoint 4, mul (x=4, y=3) at codepage2.c:36
36              number = x;
(gdb) print $rsp
$3 = (void *) 0x7fffffffdf78
(gdb) print $rbp
$4 = (void *) 0x7fffffffdf90
(gdb)
```

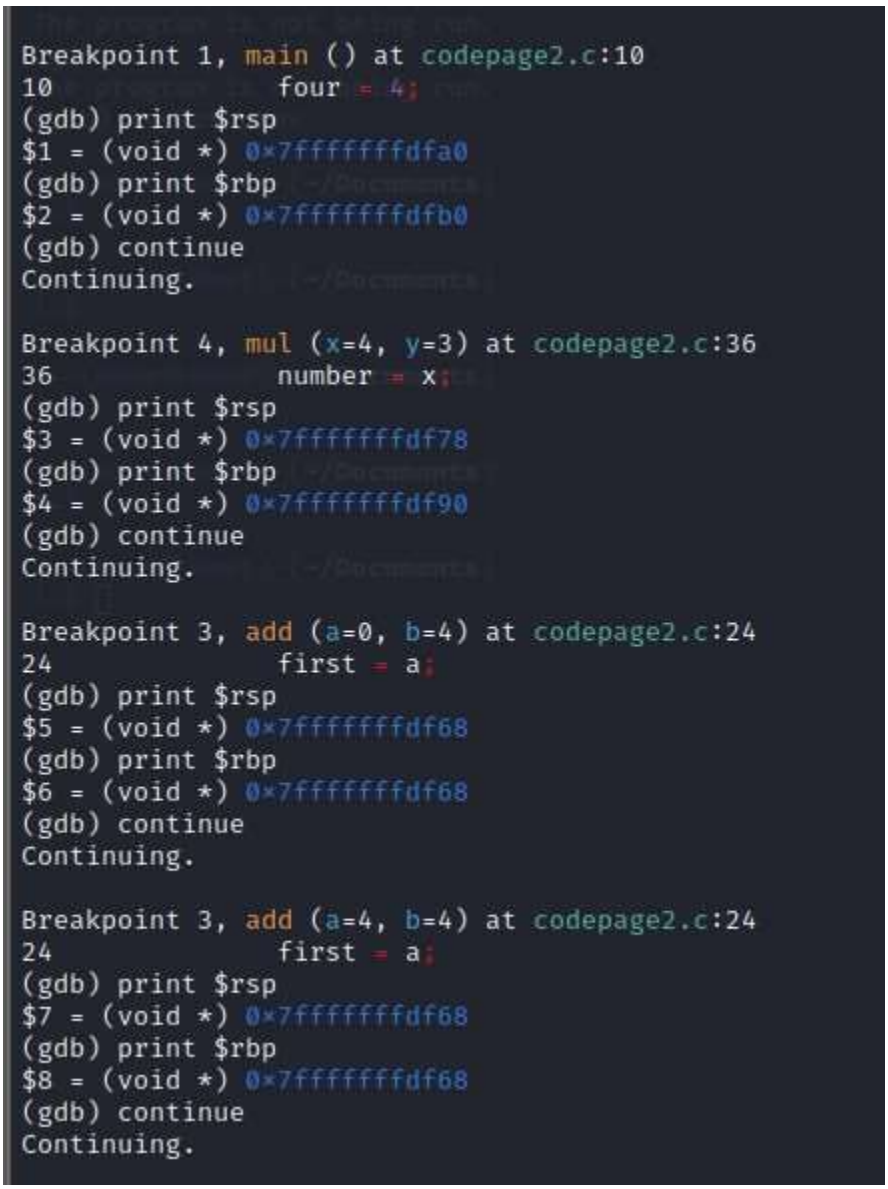
3

```
Breakpoint 3, add (a=0, b=4) at codepage2.c:24
24              first = a;
(gdb) print $rsp
$5 = (void *) 0x7fffffffdf68
(gdb) print $rbp
$6 = (void *) 0x7fffffffdf68
(gdb)
```

4

```
Breakpoint 3, add (a=4, b=4) at codepage2.c:24
24              first = a;
(gdb) print $rsp
$7 = (void *) 0x7fffffffdf68
(gdb) print $rbp
$8 = (void *) 0x7fffffffdf68
(gdb)
```

L'ensemble :



```
Breakpoint 1, main () at codepage2.c:10
10              four = 4;
(gdb) print $rsp
$1 = (void *) 0x7fffffffdfa0
(gdb) print $rbp
$2 = (void *) 0x7fffffffdfb0
(gdb) continue
Continuing.

Breakpoint 4, mul (x=4, y=3) at codepage2.c:36
36              number = x;
(gdb) print $rsp
$3 = (void *) 0x7fffffffdf78
(gdb) print $rbp
$4 = (void *) 0x7fffffffdf90
(gdb) continue
Continuing.

Breakpoint 3, add (a=0, b=4) at codepage2.c:24
24              first = a;
(gdb) print $rsp
$5 = (void *) 0x7fffffffdf68
(gdb) print $rbp
$6 = (void *) 0x7fffffffdf68
(gdb) continue
Continuing.

Breakpoint 3, add (a=4, b=4) at codepage2.c:24
24              first = a;
(gdb) print $rsp
$7 = (void *) 0x7fffffffdf68
(gdb) print $rbp
$8 = (void *) 0x7fffffffdf68
(gdb) continue
Continuing.
```

Les répétitions : Il rappel la même fonction donc il redéplace les pointeurs au même endroit, d'où les valeurs identiques.

# 4 Race-condition

Pour cet exercice nous allons exploiter un programme souffrant de failles de type *race-condition*. Vous trouverez le binaire correspondant à cet exercice sur Moodle: *print_file*

Avant de tenter un exploit, vous devez réaliser les tâches suivantes:

1. Donnez à l'utilisateur le droit d'exécuter le binaire avec les mêmes droits que *root* (à l'aide des permissions *suid*)

```
┌──(root💀host)-[/home/user/Documents/exfinal]
└─# chmod u+s print_file
```

--> et u+x

Le propriétaire du fichier doit être « root ».

sudo chown root print_file

2. Analysez l'assembleur du binaire afin de comprendre le fonctionnement du programme

gdb print_file

disassemble main

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000000011a5 <+0>:     push   %rbp
   0x00000000000011a6 <+1>:     mov    %rsp,%rbp
   0x00000000000011a9 <+4>:     sub    $0x440,%rsp
   0x00000000000011b0 <+11>:    mov    %edi,-0x434(%rbp)
   0x00000000000011b6 <+17>:    mov    %rsi,-0x440(%rbp)
   0x00000000000011bd <+24>:    mov    -0x440(%rbp),%rax
   0x00000000000011c4 <+31>:    mov    0x8(%rax),%rax
   0x00000000000011c8 <+35>:    mov    %rax,-0x10(%rbp)
   0x00000000000011cc <+39>:    mov    -0x10(%rbp),%rax
   0x00000000000011d0 <+43>:    mov    $0x4,%esi
   0x00000000000011d5 <+48>:    mov    %rax,%rdi
   0x00000000000011d8 <+51>:    call   0x1080 <access@plt>
   0x00000000000011dd <+56>:    test   %eax,%eax
   0x00000000000011df <+58>:    jne    0x1295 <main+240>
   0x00000000000011e5 <+64>:    lea    0xe18(%rip),%rdi        # 0x2004
   0x00000000000011ec <+71>:    mov    $0x0,%eax
   0x00000000000011f1 <+76>:    call   0x1060 <printf@plt>
   0x00000000000011f6 <+81>:    lea    -0x424(%rbp),%rax
   0x00000000000011fd <+88>:    mov    %rax,%rsi
   0x0000000000001200 <+91>:    lea    0xe1b(%rip),%rdi        # 0x2022
   0x0000000000001207 <+98>:    mov    $0x0,%eax
   0x000000000000120c <+103>:   call   0x10a0 <__isoc99_scanf@plt>
   0x0000000000001211 <+108>:   mov    -0x10(%rbp),%rax
   0x0000000000001215 <+112>:   lea    0xe09(%rip),%rsi        # 0x2025
   0x000000000000121c <+119>:   mov    %rax,%rdi
   0x000000000000121f <+122>:   call   0x1090 <fopen@plt>
   0x0000000000001224 <+127>:   mov    %rax,-0x18(%rbp)
   0x0000000000001228 <+131>:   movl   $0x0,-0x4(%rbp)
   0x000000000000122f <+138>:   jmp    0x125f <main+186>
   0x0000000000001231 <+140>:   lea    -0x420(%rbp),%rax
   0x0000000000001238 <+147>:   mov    %rax,%rdi
   0x000000000000123b <+150>:   call   0x1050 <strlen@plt>
   0x0000000000001240 <+155>:   sub    $0x1,%rax
   0x0000000000001244 <+159>:   movb   $0x0,-0x420(%rbp,%rax,1)
   0x000000000000124c <+167>:   lea    -0x420(%rbp),%rax
   0x0000000000001253 <+174>:   mov    %rax,%rdi
   0x0000000000001256 <+177>:   call   0x1030 <puts@plt>
   0x000000000000125b <+182>:   addl   $0x1,-0x4(%rbp)
   0x000000000000125f <+186>:   mov    -0x18(%rbp),%rdx
   0x0000000000001263 <+190>:   lea    -0x420(%rbp),%rax
   0x000000000000126a <+197>:   mov    $0x400,%esi
   0x000000000000126f <+202>:   mov    %rax,%rdi
   0x0000000000001272 <+205>:   call   0x1070 <fgets@plt>
   0x0000000000001277 <+210>:   test   %rax,%rax
   0x000000000000127a <+213>:   je     0x1287 <main+226>
   0x000000000000127c <+215>:   mov    -0x424(%rbp),%eax
   0x0000000000001282 <+221>:   cmp    %eax,-0x4(%rbp)
   0x0000000000001285 <+224>:   jl     0x1231 <main+140>
   0x0000000000001287 <+226>:   mov    -0x18(%rbp),%rax
   0x000000000000128b <+230>:   mov    %rax,%rdi
   0x000000000000128e <+233>:   call   0x1040 <fclose@plt>
```

Qu'observe-t-on ?
La vérification accès (access@alt) se fait avant le scanf :

```
call   0x1080 <access@plt>
test   %eax,%eax
jne    0x1295 <main+240>
lea    0xe18(%rip),%rdi        # 0x2004
mov    $0x0,%eax
call   0x1060 <printf@plt>
lea    -0x424(%rbp),%rax
mov    %rax,%rsi
lea    0xe1b(%rip),%rdi        # 0x2022
mov    $0x0,%eax
call   0x10a0 <__isoc99_scanf@plt>
```

Access vérifie si la personne qui a lancé le programme à bien le droit de lancer le fichier.
Dans notre exercice, c'est bien le cas.

Il y a une faille car, la fonction accès, qui a pour but de vérifier les accès, a déjà eu lieu avant le scanf. Il n'y a donc plus de vérification.

Comment ce programme aurait-il dû être codé, pour éviter ce problème ?

« Access » aurait dû être juste avant « open ».

3. Profitez de la faille *race condition* pour afficher le contenu du fichier */etc/shadow*

Droits requis :

```
-rwsrwxrwx 1 root root 16976 Oct 17 18:45 print_file
```

Exécution exemple du script :

```
┌──(user㉿host)-[~/Desktop]
└─$ ./print_file test
Number of lines to display?: 20.
aaa
bb
```

**Accéder à /etc/shadow**

De base, ça ne fonctionne pas :

```
┌──(user㉿host)-[~/Documents/exfinal/aaa]
└─$ ./print_file /etc/shadow
Permission denied
```

Rappel : on sait que notre programme passe la fonction « access »

On va utiliser une faille dans le programme pour créer un lien symbolique.
On relance le programme, avec avec un fichier texte random :

```
┌──(user㉿host)-[~/Documents]
└─$ touch final.txt
```

Puis on lance le programme :

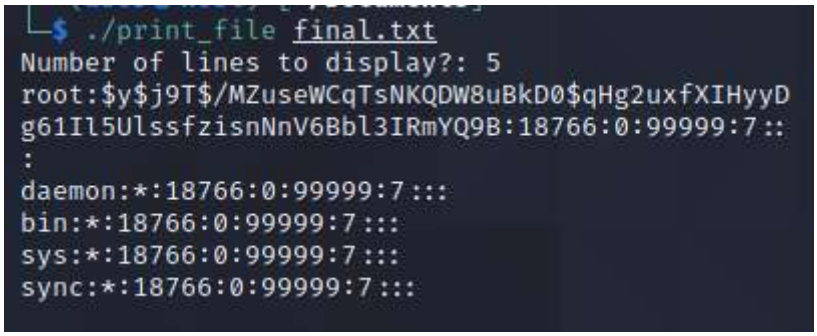
```
└─$ ./print_file final.txt
Number of lines to display?: 5
```

Sur un autre terminal, on supprime le fichier :

```
┌──(user㉿host)-[~/Documents]
└─$ rm final.txt
```

Puis on créé un lien symbolique :


On retourne ensuite sur notre premier terminal on donne le nombre de ligne et ca fonctionne :




Cependant, nous n'avons pas de droit d'accès sur /etc/shadow !