# Stack buffer overflow

26 juin 2025

**Objectif** : Comprendre le principe de l'attaque stack buffer overflow à l'aide d'un debugger

**Moyens** :

- VM linux (Ex : kali)

## Table des matières

# 1 Objectif

Cette activité va mettre en avant une des plus vielles attaques logiciel : le stack buffer overflow. Pour comprendre et interpréter son fonctionnement, vous utiliserez le debbuger `gdb` sur votre Vm `kali`

**Précision :** Les exemples que nous allons exploiter sont des cas d'écoles. Ils ont pour objectif de vous sensibiliser à cet type d'exploit.

# 2 Contourner l'authentification

Vous avez à disposition dans le répertoire ressource un programmable exécutable nommé : `ex1`. Il a été compilé sur Kali et ne s'exécute donc pas sur windows. Pour les besoins des tests, vous avez aussi le code source ex1.cpp dans lequel les informations ont été cachées.

Il a été compilé à l'aide du code source présent en annexe . Pour le compiler, il a fallut taper la ligne suivante dans une console

```
g++ ex1.cpp -g -o ex1
```

avec

- `g++` : utilisation du compilateur gcc
- `-g` : utilisation des informations de debug
- `-o ex1` : la sortie (le fichier exécutable créé s'appellera ex1).

Pour l'exécuter, il suffit de taper dans un terminale :

```
./ex1
```

**Votre objectif est de trouver la phrase qui s'affiche quand l'authentification est correcte** Vous allez pour cela exploiter 3 méthodes différentes pour arriver à vos fins.

## 2.1 GDB et overflow

### 2.1.1 Explication

La 1re méthode va être basée sur le débugger `gdb` et l'exploitation d'un **buffer overflow**. Vous trouverez ci-dessous quelques définitions concernant les outils et ce type d'attaque.

**Debugger**

Un débeugueur en français ou debugger en anglais est un logiciel qui aide un développeur à analyser les bugs d'un programme. Pour cela, il permet d'exécuter le programme pas-à-pas, d'afficher les informations du programme, de mettre en place **des points d'arrêt** sur des conditions ou sur des lignes du programme...

**Point d'arrêt**

Un point d'arrêt est une marque utilisée par un débugger pour arrêter un programme sur une ligne précise. Il est alors facilement possible de faire un dump mémoire (visualiser la mémoire), regarder l'état du programme....

`gdb` pour GNU Debugger est le débogueur standard du projet GNU. Il est portable sur de nombreuses distributions linux. Il s'utilise en ligne de commande. Vous verrez au cours de l'année qu'il existe sur le même principe des debuggers en mode graphique inclus dans les IDE.

**Buffer overflow**

Un dépassement de tampon ou débordement de tampon (en anglais, buffer overflow ou BOF) est un bug par lequel un processus, lors de l'écriture dans un tampon, écrit à l'extérieur de l'espace alloué au tampon, écrasant ainsi des informations nécessaires au processus.

La mémoire dans de la pile d'exécution si l'on regarde que les 3 premières variables est

| char mdp[10] | | | | | | | | | | char login[10] | | | | | | | | | | int authe. | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | a | a | a | a | a | a | a | p | p | p | p | p | p | p | p | p | p | 1 | 2 | 3 | 4 |
| *10 octets* | | | | | | | | | | *10 octets* | | | | | | | | | | *4 octets* | | | |

Si l'on fournit un login de plus grande taille que la taille allouée en mémoire, nous allons écrire dans l'emplacement réservé à la variable `authentification`.

**Exemple :** Le tableau `login` est constitué de 10 caractères. Si l'on rentre 14 fois le caractère 'Z', voila ce qu'il se passe.

| char mdp[10] | | | | | | | | | | char login[10] | | | | | | | | | | int authe. | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | a | a | a | a | a | a | a | Z | Z | Z | Z | Z | Z | Z | Z | Z | Z | Z | Z | Z | Z |
| | | | | | | | | | | ASCII (représentation hex) | | | | | | | | | | 0x5A | 0x5A | 0x5A | 0x5A |
| | | | | | | | | | | ASCII (représentation dec) | | | | | | | | | | 90 | 90 | 90 | 90 |
| | | | | | | | | | | | | | | | | | | | | *LSB* | | | *MSB* |
| | | | | | | | | | | Valeur *authentification* | | | | | | | | | | 1 515 870 810 | | | |

**Voici quelques commandes utiles pour gdb** Voici les principales commandes qui vous seront utiles.

- Lancer (ouvrir) le programme avec `gdb`

```
$ gdb ex1
```

- Visualiser le code source C dans GDB si on dispose des codes sources

```
(gdb) list
```

- Dans gdb, poser un point d'arrêt à la ligne `ligne` du fichier ex1.cpp

```
(gdb) break ex1.cpp:ligne
```

- Mettre un point d'arrêt sur une ligne de code asm.

```
(gdb)break *0x401234
```

- Visualiser l'ensemble des breakpoints

```
(gdb)info breakpoints
```

- Effacer un breakpoint

```
(gdb)delete Num_break
```

- Dans gdb, lancer le programme (qui s'arrête au prochain point d'arrêt ou a un cin)

```
(gdb)run
```

- Lance l'exécution du programme et s'arrête au début du main (permet de poser des breakpoints après que les segments soient mappés en mémoire)

```
(gdb)start
```

- Désassembler une fonction (main par exemple)

```
(gdb) disassemble main
```

- Dans gdb, visualiser la mémoire à partir de la fonction en cours (dans lequel vous avez mis un point d'arrêt)

```
(gdb) x/32x $sp
```

- Dans gdb, continuer après un point d'arrêt

```
(gdb) continue
```

### 2.1.2 A vous

1. Lancer gbd avec l'exécutable ex1

```
gdb ex1
....
(gdb)
```

2. Lancer le programme avec arrêt au début du main

```
(gdb) start
Temporary breakpoint 1 at 0x11f6: file ex1.cpp, line 9.
Starting program: /home/kali/Desktop/stackoverflow/ressource_eleve/ex1/ex1
....
```

3. Déassembler le main et poser un breakpoint sur une ligne qui vous semble intéressante par exemple avant [1]
   - `std::cout << "authentification AVANT "  << authentification << "\n";`

```
(gdb) disassemble
...
0x565562ad <+212>: push %eax
0x565562ae <+213>: call 0x56556060 <
    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
0x565562b3 <+218>: add $0x10,%esp

..
```

et

```
(gdb) break *0x565562ae
```

4. Continue alors l'exécution du programme en remplissant le login et le mot de passe

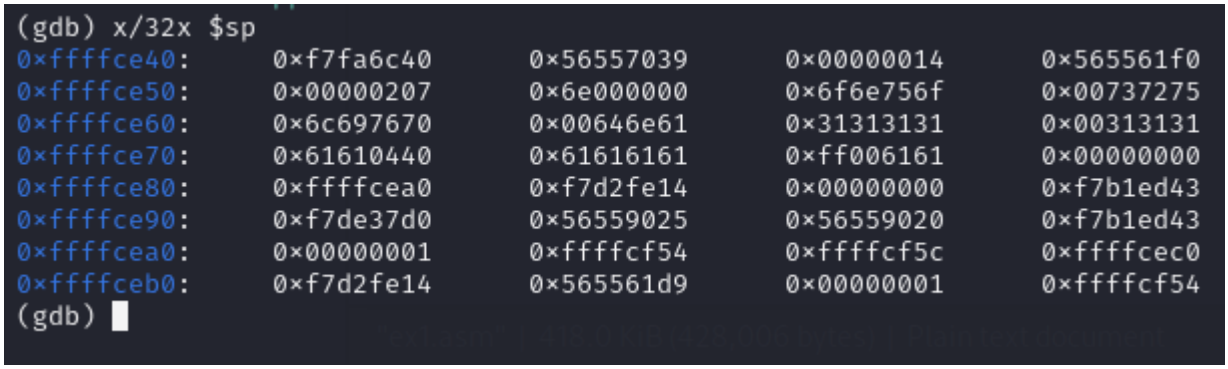
```
(gdb) c
Continuing.
Login : aaaaaaaa
Mot de passe : 11111111

Breakpoint 2, 0x565562ae in main () at ex1.cpp:24
24 in ex1.cpp
```

Vous vous êtes normalement arrêté au breakpoint.

5. Analyser alors l'état de la mémoire (la pile)

1. Par expérience, il est possible d'identifier le cin et cout dans le code assembleur avec respectivement des call à des fonctions du type : *basic_istream* et *basic_ostream*




6. Faire différents tests pour retrouver l'emplacement de la variable `authentification`. (Les commandes `run` et `x/32p $sp` vous seront très utiles). Penser à la taille des tableaux présent dans le code source.

7. Une fois que vous avez identifié l'emplacement en mémoire des différentes données, il va falloir écrire une valeur précise dans `authentification`. Vous avez ci-dessous des éléments d'aide :

- Dans le même répertoire mais dans un autre terminal, créer une chaine de caractères (payload) contenant des caractères ASCII et des valeurs hexadécimales (non ASCII). Dans l'exemple ci-dessous, on écrit 2 caractères `'p'` puis 4 octets ayant pour valeur `\xEF\xBE\xAD\xDE` (sans correspondance hexa) dans le fichier `data_input.txt`

```
$ echo $(printf 'pp\xEF\xBE\xAD\xDE')>data_input.txt
```

- Dans gdb, lancer le programme avec sur la sortie standard les données du fichier précédemment créé (pense a ajouter un breakpoint pour observer le résultat)

```
(gdb) run < data_input.txt
```

- Vous pouvez aussi utiliser ce fichier directement dans un terminal.

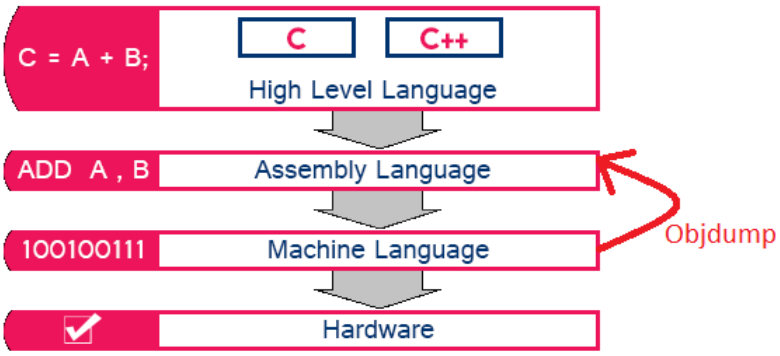
```
$ cat data_input.txt |ex1
```

Normalement en analysant et en modifiant la chaine, vous avez réussi à écraser le contenu de la variable `authentification` et donc vous avez affiché la phrase mystère.

8. Propose 2 moyens très simples d'empêcher cette attaque. Les mettre en place et les tester.

## 2.2 Code assembleur

Pour rappel, la compilation d'un programme C/C++ se passe de la manière suivante.




C'est le fichier 'langage machine' qui est exécuté sur l'ordinateur. Ce fichier est presque incompréhensible pour un humain. Cependant, il existe une possibilité de passer du langage machine au langage l'assembleur à l'aide de la commande `objdump`

Elle s'utilise de la manière suivante (dans un terminal)

```
$ objdump -D ex1 > ex1.asm
```

Vous désassemblez le fichier binaire (exécutable) `ex1` et vous mettez le résultat dans un fichier (texte) nommé `ex1.asm`. Il suffit alors de l'afficher dans un éditeur de texte de votre choix : `vim` par exemple.

Le contenu du fichier `ex1.asm` contient l'intégralité du code en langage assembleur (langage proche de la machine).

```
187 000011d9 <main>:
188     11d9:       8d 4c 24 04             lea    0x4(%esp),%ecx
189     11dd:       83 e4 f0                and    $0xfffffff0,%esp
190     11e0:       ff 71 fc                push   -0x4(%ecx)
191     11e3:       55                      push   %ebp
192     11e4:       89 e5                   mov    %esp,%ebp
193     11e6:       53                      push   %ebx
194     11e7:       51                      push   %ecx
195     11e8:       83 ec 30                sub    $0x30,%esp
196     11eb:       e8 f0 fe ff ff          call   10e0 <__x86.get_pc_thunk.bx>
197     11f0:       81 c3 04 2e 00 00       add    $0x2e04,%ebx
198     11f6:       c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
199     11fd:       c7 45 d8 70 76 69 6c    movl   $0x6c697670,-0x28(%ebp)
200     1204:       c7 45 dc 61 6e 64 00    movl   $0x646e61,-0x24(%ebp)
201     120b:       c7 45 cf 6e 6f 75 6e    movl   $0x6e756f6e,-0x31(%ebp)
202     1212:       c7 45 d3 6f 75 72 73    movl   $0x7372756f,-0x2d(%ebp)
203     1219:       c6 45 d7 00             movb   $0x0,-0x29(%ebp)
204     121d:       83 ec 08                sub    $0x8,%esp
205     1220:       8d 83 14 e0 ff ff       lea    -0x1fec(%ebx),%eax
206     1226:       50                      push   %eax
207     1227:       8b 83 e8 ff ff ff       mov    -0x18(%ebx),%eax
208     122d:       50                      push   %eax
209     122e:       e8 2d fe ff ff          call   1060 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
```

**Il ne vous est pas demandé de programmer en assembleur mais de savoir interpréter le cheminement des données.**

9. Dans le code assembleur, trouvez le login et le mot de passe permettant d'afficher la phrase mystère.

### Aide

- La structure du code assembleur est similaire à la structure du programme C++ ⇔ si des variables sont initialisées au début du main dans le code source C++, vous allez retrouver cette étape au même endroit dans le fichier assembleur.
- Les variables sont initialisées en hexa.
- Le code ASCII est utilisé pour coder les caractères.
- Le login est `pviland`

**Remarque** Des outils tels que **Ghidra** permettent une analyse plus précise et plus poussée que la commande `objdump`.

### 2.3 Visualisation de variable

La commande `strings nomProgramme` extrait et affiche toutes les chaînes de caractères ASCII lisibles contenues dans un fichier binaire, souvent un exécutable.

10. En utilisant cette commande, trouver le login et le mot de passe
11. Proposer une solution pour éviter cette fuite.

## 3 Exécution d'une fonction non utilisée

Vous allez voir qu'il est possible d'aller plus loin dans l'exploitation du buffer overflow mais l'attaque est plus compliquée. Vous allez pouvoir exécuter des fonctions qui normalement ne sont pas censées être exécutées. Une protection pour ce type d'attaque (ASLR) est par défaut mise en place sur Kali, vous allez la désactiver.

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

*Il est possible de passer outre cette protection mais l'attaque devient plus compliquée à mettre en place.*

**A la fin de l'activité, il faudra remettre réactiver la protection**

```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

**Objectif :** Le but est **d'exécuter la fonction** `fctNoUse` pour afficher le message caché [2]. le programme est présent en annexe **??**. Ce n'est pas grave s'il ne se termine pas correctement.

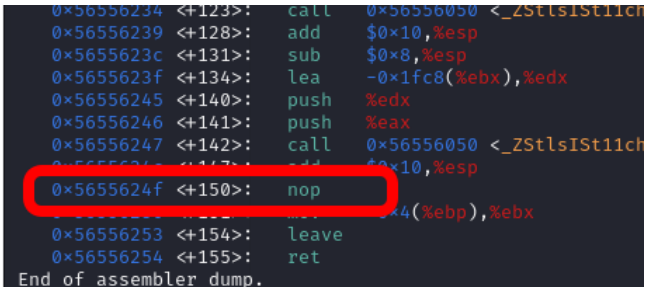
Pour cela, vous avez à disposition 2 fichiers

- Un fichier exécutable sous kali `ex2` compilé avec très peu d'informations de debug.
- Le code source partiel.

**Respecter les étapes ci-dessous et vous découvrirez le message** Vous utiliserez les commandes de debug que vous avez utilisées plus haut.

1. Etudiez le code source fournit et comprenez son fonctionnement.
2. Deassemblez le programme et afficher le résultat. Et
3. Lancez le programme avec gdb
4. Lancer le débogage avec `start`
5. Mettez un breakpoint à la fin de la fonction `fctUtile`. Pour cela, vous allez utiliser l'adresse de la ligne sur laquelle vous souhaitez vous arrêtez.Par exemple, si vous souhaitez vous arrêter à la ligne en rouge (ce qui est normalement le cas)

2. Aide : c'est une situation de Quality land (Marc-Uwe Kling).

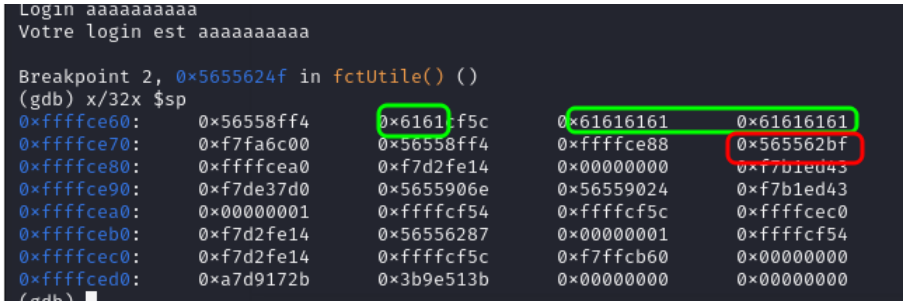



Vous devez dans `gdb` taper la commande suivante :

```
(gdb) break *0x5655624f
```

6. Continuer l'exécution avec `c` ou `continu`
7. Lorsque vous arrivez au breakpoint, visualiser la mémoire.

Vous devez avoir un résultat très similaire.




- En vert : les données rentrées par l'utilisateur en hexa. Dans mon cas : `aaaaaaaaaa`
- En rouge : l'adresse de retour de la fonction.

8. Quelle est l'utilité de l'adresse de retour (en rouge) ? Vous pouvez regarder où se situe cette adresse dans le fichier asm (en enlevant le 40). parlez en à votre très cher professeur.
9. Une fois que vous avez compris l'utilité de cette adresse, préparez votre payload et injecter la dans le programme : utilisez les commandes vues en début d'activité. **Quelle est la phrase mystère**.

**PENSER à réactiver la protection**

```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

# 4 Conclusion

Vous êtes des grands spécialiste sdu **Buffer overflow** et de **l'assembleur**. La finalité pour contrer cela : utiliser toujours des fonctions qui analyse les données de l'utilisateur et ne jamais mettre des mots de passe en claire dans un exécutable.

# A Code source ex1

```
#include <cstdio>
#include <cstring>
#include <iostream>



int main( void )
{
 int authentification= 0;
 char login[10];
 char mdp[10];

 char monLog[] = "_______";
 char monMdp[] = "________";


  std::cout << "Login : ";
 std::cin >> login;

 std::cout << "Mot de passe : ";
 std::cin >> mdp;


 std::cout << "authentification AVANT " << authentification << "\n";

 if( std::strcmp(login, monLog) == 0 && std::strcmp( mdp, monMdp) == 0 )
 {
  std::cout << "Verification OK" << "\n";
  authentification = 1;
 }
 std::cout << "authentification APRES " << authentification << "\n";

 if( authentification )
 {
  std::cout << "Acces autorise\n";
  std::cout << "_____________________________________
  _____________________________________________\n" ;
 }
 else
 {
  std::cout << "Acces non autorise\n";
 }

 return ( 0 );
}
```

# B Code source ex2

```
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

char phrase[] ="_____________________________________________";

void fctUtile()
{
 char login[10];
 cout << "Fct utile \n";
 cout << "Login ";
 cin >> login;

 cout << "Votre login est " << login << "\n";


}

void fctNoUse()
{
 cout << phrase;
}


int main( void )
{
 cout << "Debut\n";
 fctUtile();
 cout << "fin\n";
 return ( 0 );
}
```