esiea
ECOLE D'INGENIEURS
DU MONDE NUMERIQUE

RAPPORT
Exploitation de vulnérabilités Linux

Vincent Drouin
vdrouin@et.esia.fr

SOMMAIRE

1] LE PROGRAMME....3

2] LES VULNERABILITES....3

3] L'ETAT DE LA STACK....4

4] L'EXPLOIT....5

5] LIMITATIONS....6

1] Le programme

Le programme est un interpréteur de commandes via socket, qui expose une fonctionnalité de décompression.
La fonction doprocessing dispose d'un tableau de 1024 octets.

- La commande 0 permet d'envoyer un nombre d'octets de 1 à 255 par la socket, pour les faire écrire à la position du curseur.
  Le curseur est incrémenté du nombre d'octets écrits.
- La commande 1 permet de copier une partie du tableau et d'écrire la copie à partir de la position du curseur.
  La copie se fait à partir d'une valeur d'offset spécifiée, et selon une taille limitée. On peut également spécifier le nombre de fois où l'opération de copie sera répétée.
  Le curseur est incrémenté du nombre d'octets écrits.
- La commande 0xff permet de lire la valeur du curseur, ainsi que le contenu du tableau et renvoyer le résultat par la socket.

2] les vulnérabilités

La vulnérabilité trouvée dans le code se situe dans la commande numéro 1.
On vérifie bien que l'offset est inférieur au curseur pour éviter une erreur d'overflow quand on calcule "cursor- offset".
En revanche, rien n'empêche d'utiliser une taille de copie supérieure à la valeur de l'offset, ce qui va nécessairement engendrer la manipulation de données non initialisées.

De plus on vérifie qu'on dépasse la taille maximale du tableau de manière non sécurisée : la valeur du curseur avant le memmove n'indique absolument pas si "cursor + size > 1024".
Donc on peut écrire dans la mémoire en dehors du tableau.

Pour corriger ces vulnérabilités, j'ajouterai des contrôles pour vérifier que :

- la taille demandée est bien inférieure ou égale à l'offset (avant la boucle)
- la taille + le curseur ne dépassent pas les limites du tableau (avant chaque appel à memmove)

3] l'état de la stack

La constante WELCOME_MESSAGE est présente dans la stack, autour de l'adresse de base du programme (base_chall_address + 0x1128).

Dans la fonction doprocessing:

rbp- 8 * 1027 = command
rbp- 8 * 1026 = tmp
rbp - 8- 8 * 1024 = output_buff
rbp- 8 = cookie
rbp
rbp + 8 = rip

Les variables entières utilisées (cursor, size, offset…) sont localisées en dessous de command dans la
stack.

4] l'exploit

L'exploit se fait en plusieurs temps :

1. récupération de l'adresse de base de la libc, à partir d'un leak dans la première partie du tableau output_buff, avec une recopie non initialisée à l'index 0.
2. récupération de valeurs intéressantes à partir d'un leak dans les 255 octets situés "après" le tableau (au-dessus dans la stack), toujours grâce à une recopie de données non initialisées.
    a. cookie de session (offset de 8 octets après output_buff dans les traces)
    b. rbp (offset de 16 octets après output_buff)
    c. rip (offset de 16 octets après output_buff)
    d. adresse de base du programme chall (calculé à partir du rip)
3. préparation des valeurs utilisées pour l'exploit :
    a. adresse de l'instruction "pop rdi" (obtenue avec radare2)
    b. adresse de la fonction system (offset calculé via gdb)
    c. adresse de la fonction exit (offset calculé via gdb)

Le code vise à remplacer la plage de 255 octets après le tableau, pour remplacer la valeur de rip par un code qui va permettre l'exécution de code arbitraire.

Pour effectuer l'écrasement du rip, j'ai de nouveau utilisé la commande 0, pour envoyer les informations souhaitées dans les 255 dernières entrées du tableau, puis la commande 1, pour recopier cette plage mémoire juste après le tableau.

La payload est placée au début du tableau output_buff, dont on connaît l'adresse grâce au leak, et grâce à une analyse via gdb en la recalculant à partir de la valeur de saved rbp : addresse payload = valeur saved rbp - 0x510.

Les instructions passées sont les suivantes :

- @rip : pop rdi, ret
- @rip+8 : @payload (= adresse output_buff)
- @rip+16 : @system dans la libc
- @rip+24 : @exit dans la libc

La commande en payload, une fois exécutée par la fonction system, crée un reverse shell dans le répertoire tmp.

5] Limitations

L'exécution de ce code a très bien fonctionné sur ma machine virtuelle Ubuntu tournant en live disk sur ma debian.
En revanche, une machine virtuelle Ubuntu similaire mais installée sur mon Mac personnel donne des problèmes avec la libc, tant au niveau de l'emplacement des adresses de la libc dans le leak qu'au niveau des offsets de exit() et system() par rapport à l'adresse de base de la libc.

Un peu plus de temps m'aurait donc permis d'être plus rassuré concernant la pertinence du leak choisi pour calculer l'adresse de la libc, et de comprendre les différences d'offset dans la libc.
Je n'ai pas pu confirmer l'hypothèse que des versions différentes de la libc sont probablement utilisées.