TP 2 - Exploitation Vulnérabilités Logicielles – part 2

Florent Autréau

2016-2017

Encadrants :
Autreau Florent : florent.autreau@imag.fr

Le compte de la machine virtuelle est securimag / 123456

1. Épreuve d'exploitation logicielle (buffer overflow) :

Le programme à exploiter se nomme bof, et se trouve sur le bureau.
Nous allons ici voir en détail la procédure de découverte du bug et la méthodologie qui mène à son exploitation.

1) Lancer le programme dans un terminal.

2) Lancer avec une grande entrée (voir en annexe « Utilisation de python » pour générer facilement des entrées). L'objectif est d'obtenir un crash.

3) Ouvrir le programme dans IDA et étudier le code. En s'aidant juste de IDA, ou se trouve le problème à votre avis ? (indice : chercher des fonctions de copie)

4) Utiliser gdb sur le programme (voir en annexe « Utilisation de gdb »). Lancer le avec une entrée qui produit un crash. Qu'est-ce qui provoque un crash ? Indice : que vaut le registre EIP ? A quoi sert ce registre ?

5) Vous pouvez vous servir de gdb pour voir l'évolution du programme. Rajouter un breakpoint avant l'appel à la fonction « vuln » et à sa sortie. Qu'observez vous (ou le crash se produit?) ? Utilisez les commandes fournies en annexe pour vous familiarisez avec gdb (voir l'évolution de EIP, placer des breakpoints, regarder l'état de la pile...)

6) On peut planter le programme avec une longue chaîne de caractères, cependant il est intéressant de trouver quelles sont les octets en entrée qui permettent de contrôler directement EIP.
L'idée est donc de trouver une entrée de la forme :
[PADDING] [ADDR EIP]
Pour trouver la taille du padding, vous pouvez procéder par dichotomie, en mettant un padding rempli d'un caractère (par exemple de « A »), puis un autre caractère pour la valeur réécrite pour eip : 'A' * n + 'BBBB'
Il vous faut donc trouver la valeur de n pour lquelle EIP prendra BBBB en valeur.
Vous pouvez aussi utiliser le script python « pattern.py ». Il permet de générer une longue chaîne de caractères sans répétition.

7) Vous avez maintenant la possibilité de contrôler le flot d'exécution en contrôlant la valeur que prendra le registre EIP. L'idée à présent est de faire

exécuter au programme un code pour lequel il n'a pas été conçu. Pour ce faire on peut utiliser ce qu'on appelle un shellcode. Un shellcode est un petit bout de programme écrit en assembleur. Vous pouvez en trouvez de nombreux sur Internet (exemple : http://shell-storm.org/shellcode/). Vous pouvez par exemple utiliser celui ci : http://shell-storm.org/shellcode/files/shellcode-827.php qui lance un shell /bin/sh.
Si le shellcode se trouve dans l'entrée du programme, il sera copié dans le tableau où le buffer overflow se produit, et il se trouvera ainsi sur la pile d'exécution du programme.
L'idée alors est de rediriger le flot d'exécution directement sur l'emplacement de la pile du programme ou se trouve le shellcode.
Il nous faut donc d'abord trouver l'adresse ou se trouvera notre shellcode en mémoire. Pour la trouver, on peut mettre un breakpoint après l'appel à strcpy, et inspecter la pile (rappel : sous gdb x/16x $esp affiche les 16 premières valeurs de la pile).
N'oubliez pas, nous sommes en little-endian, si vous voulez avoir ABCD comme adresse il faudra l'écrire sous la forme DCBA en entrée
Faites un schéma expliquant le positionnement de votre entrée par rapport à la pile et expliquant ce se qui passe durant l'overflow.
Construisez l'exploit et lancer le sous gdb.

Astuce : pensez à utiliser l'aide python fournie en annexe pour construire votre exploit

8)
Vous remarquerez que l'exploit fonctionne sous gdb, mais ne fonctionne pas en dehors de gdb, ceci est dû au fait que gdb modifie légèrement l'environnement dans lequel le programme est lancé. Ainsi l'adresse du shellcode que vous trouvez grâce à gdb sera un peu différente dans une exécution normale. Il peut être parfois difficile de trouver la valeur exacte de cette adresse. L'astuce ici est simplement de rajouter un ensemble d'instructions qui ne font rien (NOP en assembleur, qui correspond à l'opcode 0x90) avant le shellcode (200 NOP par exemple). Il suffira alors de rediriger le flot d'exécution sur l'espace rempli de NOP qui précède le début de notre shellcode. Ainsi nous n'avons pas à connaître l'adresse exacte du début du shellcode.
Pour trouver une adresse dans le padding NOP, vous pouvez procéder comme dans l'étape précédente (eg : mettez un breakpoint après l'appel à strcpy, et chercher dans la pile une zone remplie de NOP).
Faites un schéma expliquant le positionnement de votre entrée par rapport à la pile avec le NOP Padding
Construisez et lancez ensuite l'exploit.

Annexe :

IDA :

Pour lancer IDA, dans un terminal « ida », ou lancer le script ida.sh se trouvant sur le bureau.

Utilisation de gdb :
x/x ADDR : affiche la valeur se trouvant à l'adresse ADDR
x/x $REG: affiche la valeur pointée par le registre (exemple : x/x $esp, affiche

un élément de la pile)
x/4x ADDR : affiche 4 octets depuis l'adresse ADDR
x/i ADDR : affiche l'octet sous forme d'une instruction
Vous pouvez combiner cette syntaxe, et par exemple afficher les 2 prochaines instructions à exécuter avec :
x/2i $eip

info registers : affiche les valeurs de tout les registres

breakpoint * ADDR : place un breakpoint sur ADDR
disas func : désassemble la fonction func

run : lance le programme
run ARG : lance le programme avec l'argument
nexti : exécute la prochaine instruction
continue : continue l'exécution (après un breakpoint)

Utilisation de python :
Vous pouvez vous servir de python pour générer des entrées à vos programmes.

Exemple d'utilisation :
./programme $(python -c ' print "A" ')
Lance le programme avec A pour argument

./programme $(python -c ' print "A"*10 ')
Lance le programme avec AAAAAAAAAA pour argument

./programme $(python -c ' print "\x41"*10 ')
Lance le programme avec AAAAAAAAAA pour argument (0x41 étant le code ASCII de A)

Vous pouvez aussi utiliser la même syntaxe pour lancer d'autres commandes, par exemple pour donner un fichier en entrée d'un programme :
./programme $(cat fichier.txt)

Vous pouvez utiliser python aussi dans gdb, exemple :
run $(python cat fichier.tx)
Vous permettra d'avoir facilement le fichier fichier.txt en entrée du programme sous gdb.