Sécurité Logicielle

– Examen (**1**) –

1 Attaque ROP (8 points)

Le programme `vulnerable` (x86-32) a un bug de type '*buffer-overflow*' et permet d'injecter des octets NULL en mémoire. On veut lancer un shell via un exploit de type ROP (*Return-Oriented Programming*). On sait qu'il faut un padding de 96 octets pour arriver à placer `0xdeadbeef` dans le registre `eip` (voir le script qui suit).On connait par ailleurs l'ensemble des gadgets qu'il contient (voir la liste suivante) et l'adresse de la section `.data` qui est localisée à `0x0804a024`.

Construisez le programme ROP qui permet de lancer un shell.

```
#!/usr/bin/env python
from struct import pack

# Padding goes here
p = 'A' * 96

p += pack('<I', 0xdeadbeef) # mnemonic arg1 arg2; ret
...

print(p)
```

```
$ ROPgadget --binary vulnerable
Gadgets information
============================================================
0x0804863c : add al, 0x24 ; ret
0x080484d8 : add al, 8 ; add ecx, ecx ; ret
0x08048471 : add al, 8 ; call eax
0x080484ab : add al, 8 ; call edx
0x0804869f : add bl, dh ; ret
0x08048388 : add byte ptr [eax], al ; add esp, 8 ; pop ebx ; ret
0x080487f3 : add dword ptr [eax], eax ; inc ecx ; ret
0x080484da : add ecx, ecx ; ret
0x08048475 : add esp, 0x10 ; leave ; ret
0x0804838a : add esp, 8 ; pop ebx ; ret
0x08048473 : call eax
0x080484ad : call edx
0x0804855b : inc eax ; ret
0x0804856e : inc ebp ; clc ; ret
0x08048567 : inc ebp ; hlt ; mov dword ptr [edi], ebx ; ret
0x08048559 : inc ebp ; in al, dx ; inc eax ; ret
0x080487f5 : inc ecx ; ret
0x0804884b : inc edi ; push cs ; adc al, 0x41 ; ret
0x08048572 : int 0x80
0x08048713 : jmp eax
0x0804869d : lea esi, dword ptr [esi] ; ret
0x08048637 : lea esp, dword ptr [ecx - 4] ; ret
0x08048478 : leave ; ret
0x08048569 : mov dword ptr [edi], ebx ; ret
0x08048562 : mov dword ptr [edx], ebx ; ret
0x0804863b : mov eax, dword ptr [esp] ; ret
0x0804853c : mov ebx, dword ptr [ebp - 4] ; leave ; ret
```

```
0x08048440 : mov ebx, dword ptr [esp] ; ret
0x0804856a : pop ds ; ret
0x08048571 : pop eax ; int 0x80
0x08048636 : pop ebp ; lea esp, dword ptr [ecx - 4] ; ret
0x0804869b : pop ebp ; ret
0x08048698 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804838d : pop ebx ; ret
0x08048554 : pop ecx ; pop edx ; ret
0x0804869a : pop edi ; pop ebp ; ret
0x08048555 : pop edx ; ret
0x08048699 : pop esi ; pop edi ; pop ebp ; ret
0x08048638 : popal ; cld ; ret
0x0804884c : push cs ; adc al, 0x41 ; ret
0x080484ff : push eax ; call edx
0x080484d6 : sub al, 0xa0 ; add al, 8 ; add ecx, ecx ; ret
0x0804846f : sub al, 0xa0 ; add al, 8 ; call eax
0x080484a9 : sub al, 0xa0 ; add al, 8 ; call edx
```

2 On the Effectiveness of Full-ASLR on 64-bit Linux (12 points)

Lisez l'article "*On the Effectiveness of Full-ASLR on 64-bit Linux*" par H. Marco-Gisbert et I. Ripoll (DeepSeC, Vienne, 2014). Puis rédigez des réponses aux questions.

Questions

1. Rappelez les principes de l'ASLR (Address-Space Layout Randomization), du SSP (Stack Smashing Protector), du NX (No-eXecute), du RELRO (RELocation Read-Only) ainsi que du PIE (Position-Independant-Executable) et contre quels types d'attaques il ont chacun été introduit.
2. Expliquez comment le mécanisme de segmentation mémoire [1] permet à une bibliothèque partagée (*shared-library*) d'être chargée une seule fois en mémoire et d'être partagée entre plusieurs processus.
3. Expliquer ce que la valeur "`offset2lib`" représente et en quoi elle est une faiblesse pour la sécurité.
4. Le serveur de démonstration décrit dans l'article utilise '`fork()`' pour relancer un nouveau processus en cas de crash. Commenter et critiquer ce choix d'implémentation, proposez éventuellement une autre approche plus sûre (et expliquez en quoi elle est plus sûre).
5. Dans le listing 3 (p.5), à quoi correspondent les adresses des instructions ? Expliquez pourquoi elles sont si basses et pourquoi les auteurs ne retiennent finalement que '`0x2df`' (sur '`0x12df`') ?
6. Dans l'étape 2 de l'attaque, expliquez comment doit être menée l'attaque brute-force sur le SSP et sur les 3 octets restants pour récupérer les bonnes valeurs. Et, aussi, pourquoi, sur les 768 possibilités, il est suffisant de n'en parcours que 392 en moyenne pour l'adresse.
7. Dans l'étape 3 de l'attaque, pourquoi masque-t-on l'adresse que l'on vient de retrouver avec `0xfff` ?
8. Donnez quelques éléments qui peuvent expliquer les variations observées pour l'`offset2lib` entre les différentes versions de la `libc` sur différentes distributions (Table 3, p.6).
9. Expliquer le principe de *Renew-SSP* et des *Random zones* (ASLRv3).
10. Donnez au moins trois façon de rendre inopérante l'attaque qui est décrite dans l'article (en prenant celles qui sont déjà suggérées dans l'article et éventuellement d'autres).
11. L'article présente un *buffer-overflow*, comment serait l'attaque si la faille était de type *format-string* ?
12. Enfin, donnez des arguments pour ou contre le patch proposé qui introduit l'ASLRv3.

1. Qui a donné le très célèbre "*segmentation fault*".