TP 2 : Exploitation de *buffer overflow*

Romain CARRÉ
romain.carre@cea.fr

L'objectif de ce TP est de suivre pas à pas les étapes d'une **exploitation de *buffer overflow***. De nos jours, et pour toutes les raisons déjà évoquées en cours (la non-exécutabilité de la pile, le mécanisme de randomisation des adresses, etc.), il est peu probable de rencontrer de telles failles dans la nature. En revanche, cet exercice présente un intérêt pédagogique sur la démarche et les techniques d'exploitation d'une vulnérabilité, ainsi que sur la manipulation des outils afférents disponibles sur les systèmes Unix.

Bien que les durées indiquées dans le titre de chacune des parties soient approximatives, elles peuvent vous donner une idée du temps attendu pour réaliser les différentes tâches mentionnées. Si vous êtes totalement coincés sur une question, ou une procédure particulière du TP, inutile d'attendre 1h avant de venir me le signaler : faites moi signe !

1 Environnement de travail (∼10min)

- 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`.
- récupérez le fichier `vuln.c` sur la clef USB de votre encadrant
- 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 (∼20min)

- parcourez le code source de `vuln.c`, et mettez en évidence la vulnérabilité.
- 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
```

- de quoi `gcc` vous prévient-il avec un *warning* ? ca vous rappelle quelque chose ?
- testez le programme normalement. décrivez chaque étape de son fonctionnement.
- vérifiez qu'il est vraiment vulnérable en provoquant une erreur de segmentation.

3 Exploitation *proof-of-concept* (∼45min)

- 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.
- 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)
- 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.
- pourquoi la valeur de ce registre est-elle utile (indice : que fait l'instruction `RET`) ?
- la valeur d'`esp` juste avant le crash peut aussi être récupérée via `dmesg`.
- o_O ? que remarquez-vous ? comment expliqueriez-vous cette différence ?

– 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)

Rappels :
- comme vous travaillez sous Linux, la convention d'appel pour les appels systèmes est basée sur une interruption, avec passage des arguments dans les registres.
- **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* ?

4 Exploitation avec évasion d'IDS (∼45min)

- 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.
- 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.
- imaginez des mécanismes plus complexes d'obfuscation de *payload*. décrivez-les.
- 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* ?
- 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 ?

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* ?

5 Correction et questions (∼60min)

6 Annexes

Commandes `gdb` usuelles :

`disas <symbol>`	`disassemble <symbol>`	désassemble à partir de `<symbol>`
`b *<addr>`	`breakpoint *<addr>`	place un *breakpoint* à l'adresse `<addr>`
`run`	`run`	lance l'exécution du programme
`i r`	`info registers`	affiche l'état complet des registres
`x/s $<reg>`	`x/FMT $<reg>`	affiche le contenu du registre `<reg>`

Instructions assembleur :

`mov eax, <val>`	`B8 <val> 00 00 00`, ou `6A <val> 58`
`mov ebx, <val>`	`BB <val> 00 00 00`, ou `6A <val> 5B`
`mov edx, <val>`	`BA <val> 00 00 00`, ou `6A <val> 5A`
`int <val>`	`CD <val>`
`lea ecx, [esp+<val>]`	`8D 4C 24 <val>`
`xor [esp+<val1>], <val2>`	`83 74 24 <val1> <val2>`