TD/TP5 - Exploitation d'un débordement de tampon (*buffer overflow*)

Vous effectuerez ce TD/TP sur la machine virtuelle Linux / CentOS.

À la fin de chaque séance, pensez à sauvegarder votre travail si nécessaire.

Les programmes sont disponibles sur https://gitlab.com/Thomas-Hugel/IUT/PPN-2013/M2101-Bas-Niveau

Comme vous l'avez vu dans le TD/TP précédent, avec le langage C il est facile de déborder d'une chaîne de caractères de taille fixe (ce qu'on appelle également *tampon*). Le terme anglais est *buffer overflow*. Nous allons voir maintenant comment l'utilisateur peut exploiter cette faille pour détourner le cours normal de l'exécution de ce programme vers un morceau de code arbitraire.

Au préalable, vous devez désactiver une sécurité du système d'exploitation (consistant à introduire de l'aléatoire dans les adresses, pour qu'elles changent à chaque exécution) :

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

1 Fabrication de chaînes de caractères en Bash

1. Lisez dans le manuel le rôle des options `-n` et `-e` de la commande `echo`. Lancez ensuite la commande suivante, et interprétez-en le résultat :

   ```
   echo -n -e "\x41\x42\x43"
   ```

2. Vous allez devoir répéter un caractère donné un certain nombre de fois. Exécutez les commandes suivantes et comprenez ce qui se passe :

   ```
   eval printf "%sA" {1..20}
   eval printf "%.sA" {1..20}
   eval printf "%.s'\x41'" {1..20}
   ```

2 Constatation du débordement

1. Récupérez le programme `BufferOverflow.c` :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6          char buffer[100];
7          if (argc > 1) {
8                  strcpy (buffer, argv[1]);
9                  printf ("Voici votre saisie: %s\n", buffer);
10         }
11         return 0;
12 }
```

2. Lisez-le et identifiez l'erreur de programmation. Ne la corrigez pas, nous allons l'exploiter.
3. Compilez-le de la façon suivante (cela va désactiver une autre sécurité - à savoir rendre la pile exécutable - afin de nous faciliter le travail) :

   ```
   gcc -g -z execstack BufferOverflow.c -o BufferOverflow
   ```

4. Exécutez-le en lui passant successivement `AAAAAAAAAA` (noté A^{10}), A^{100} et A^{1000}.

3 Analyse du débordement

1. Déboguez l'exécution avec A^{1000} étape par étape, et regardez où l'erreur survient.
2. Déboguez l'exécution avec A^{10} et regardez ce qui se passe là où il y avait l'erreur. Notez en particulier le nom et l'adresse de la fonction dans laquelle on arrive.
3. Recommencez l'exécution avec A^{99} :
   (a) arrêtez-vous à la ligne 11 ;
   (b) contemplez la pile :
   ```
   x/20gx $rsp
   ```
   (c) quelle est l'adresse de `buffer` ?
   (d) à quelle adresse semble se trouver l'adresse de retour de la fonction `main` ?
   (e) pour être sûr qu'il s'agit bien de l'adresse de retour, regardez quelle est l'adresse du sommet de la pile lorsque l'instruction machine `ret` (à la fin du `main`) est exécutée.
4. Sur les 64 bits d'adresses, seuls 48 sont utilisés, et la moitié haute des adresses est réservée au système. Quel est l'intervalle des adresses valides pour un programme de l'utilisateur ?
5. Recommencez l'exécution avec A^{1000} :
   (a) allez à la fin du `main`, et regardez ce que vaut l'adresse de retour (au sommet de la pile juste avant `ret`) ;
   (b) cette valeur est-elle dans l'intervalle des adresses valides ?
   (c) que se passe-t-il donc au moment d'exécuter `ret` ?

4 Contrôle de l'adresse de retour

1. Vous allez d'abord écrire une adresse plausible :
   (a) en utilisant vos observations de la partie 3, déterminez le nombre p d'octets qui séparent l'adresse de `buffer` de l'adresse de retour de la fonction `main` ;
   (b) déboguez l'exécution avec A^pB^6 ; vérifiez que le contenu du cadre de `main` correspond à ce que vous aviez prévu ;
   (c) exécutez `ret` ; cette fois cela devrait fonctionner ;
   (d) que contient maintenant `rip` ?
   (e) avancez d'une instruction machine.
2. Toujours dans le débogueur, écrivez maintenant l'adresse de `buffer` à l'emplacement de l'adresse de retour (indication : utilisez `'\xff'` pour désigner le caractère de code hexadécimal `ff`). Il suffira ensuite de remplir le tampon avec les instructions voulues.

5 Shellcode

1. Observez le code assembleur de la partie 1 du TD/TP précédent. Le code machine est visible au milieu en hexadécimal. Combien de caractères nuls y a-t-il dans ce code ?
2. Le problème avec les caractères nuls, c'est qu'ils servent à délimiter en C la fin des chaînes de caractères. Vous pouvez donc mettre dans votre tampon du code arbitraire, mais pas n'importe quel code : ce code ne doit pas contenir de caractère nul ! On appelle *shellcode* un morceau de code conçu spécialement à cet effet.
   (a) Récupérez le fichier `shellcode41498.txt`, qui a été fabriqué à partir de `https://www.exploit-db.com/exploits/41498/`. Ce fichier contient les instructions machine suivantes : devenir `root` et lancer un shell en tant que `root`. À l'aide de la commande `sha256sum`, vérifiez que sa somme de contrôle est de la forme $9da \ldots e6f$.

(b) À l'aide d'une commande appropriée, déterminez le nombre d'octets de ce fichier.

3. En utilisant le débogueur, lancez le programme `BufferOverflow` en lui passant en paramètre une chaîne consistant en :
   (a) un certain nombre de fois (à déterminer) le caractère `\x90` (qui correspond à l'instruction vide `nop`) ;
   (b) suivi du shellcode ;
   (c) suivi de A^{30} ;
   (d) suivi d'une adresse à déterminer ; cette adresse doit être comprise entre le début du tampon et le début du shellcode. L'intérêt d'avoir rajouté des `nop` au début est que vous avez droit à une certaine marge d'erreur sur l'adresse à renseigner ;
   (e) en utilisant ce que vous avez fait à la section 4, arrangez-vous pour écraser exactement l'adresse de retour de `main` avec une adresse qui va dérouter l'exécution vers le shellcode.

6 Prise de contrôle

1. Recompilez `BufferOverflow.c` avec les mêmes options que ci-dessus, sauf `-g`. Faites en sorte que l'exécutable `BufferOverflow` :
   (a) soit exécutable par tous ;
   (b) soit la propriété de `root:root` ;
   (c) ait le bit `setuid` positionné.
2. Recommencez la démarche de la partie 5 hors du débogueur. Les adresses vont changer légèrement, toutefois la distance entre le début du tampon et l'adresse de retour devrait rester la même. En tâtonnant un peu, vous devriez pouvoir trouver une adresse qui va pointer sur un `nop`.
3. Vous saurez que vous aurez réussi votre piratage lorsque la commande `whoami` vous répondra : `root`.
4. Faites-vous plaisir, puis tapez sur `Ctrl+D` pour sortir du shell privilégié.

7 Piratage en aveugle

Jusqu'à maintenant vous avez pu examiner le code source et visualiser l'exécution du programme au débogueur. Maintenant que vous avez les idées claires sur ce qui se passe, il est temps de passer à la vitesse supérieure et de craquer un exécutable sans accès au code source ni au débogueur.

1. Récupérez l'exécutable `Opaque.out` :
   (a) vérifiez que sa somme de contrôle `SHA256` est de la forme $2f4\ldots943$;
   (b) et exécutez-le avec une chaîne de caractères en paramètre. Comme vous le voyez, ce programme ressemble fort au précédent. Toutefois il y a des variables locales en plus, et la taille du tampon a changé. Vous allez donc écrire des scripts pour automatiser la recherche des paramètres de votre chaîne de caractères.
2. Auparavant, comme dans la partie 6, faites en sorte que l'exécutable `Opaque.out` :
   (a) soit exécutable par tous ;
   (b) soit la propriété de `root:root` ;
   (c) ait le bit `setuid` positionné.

3. Vous allez commencer par chercher la distance entre le début du tampon et l'adresse de retour à écraser. Pour ce faire, vous allez passer successivement à votre programme A^8, A^{16}, A^{24}… jusqu'à ce qu'il y ait une erreur de segmentation. Vous pourrez supposer qu'à la première erreur (disons, sur A^{8k}), vous avez atteint l'adresse de retour. Votre chaîne frauduleuse sera donc de la forme A^{8k} + adresse frauduleuse. Écrivez donc un script `find-buffer-size.sh` qui recherche cette longueur $8k$ et l'affiche. Notez-la précieusement.
4. Maintenant écrivez un script `find-buffer-address.sh` qui va tenter de trouver une adresse frauduleuse. Pour vous faciliter la tâche, vous remplacerez A^{8k} par un certain nombre de `nop`, le shellcode de 31 octets, puis A^{30} (comme dans la partie 5). Il reste à trouver une adresse pointant sur un `nop` : votre script tentera toutes les adresses multiples de 32 (car les adresses font 8 octets et sont alignées sur 8 octets sur la pile, et car votre shellcode fait 31 octets), en partant du haut de la pile (`0x7fffffffffff`) et en redescendant jusqu'à (`0x7fffffff0808`), ce qui sera largement suffisant (combien d'adresses y a-t-il dans cette plage *grosso modo* ?).
   Attention : vos adresses ne doivent pas contenir de caractère nul !
   Indication : il pourra vous être utile d'utiliser `printf` avec le format suivant (hexadécimal complété par des 0 en tête, sur 2 caractères) :

   ```
   $ printf "%02x" 15
   0f
   ```

   Cette fois, vous allez itérer jusqu'à ne plus avoir d'erreur de segmentation. Affichez l'adresse présumée pointer dans le tampon. Et alors, votre script ne sera pas même terminé que vous aurez gagné ! Faites-vous plaisir, puis faites `Ctrl+D` pour sortir du shell privilégié : notez qu'alors votre script reprend son exécution, affiche l'adresse trouvée, et, espérons-le, s'arrête…
5. Écrivez un petit script `exploit.sh` qui utilise `Opaque.out` pour ouvrir un shell en tant que `root` directement, sans refaire toute la recherche ci-dessus.
6. Écrivez maintenant un programme `Candidate.c` qui pourrait être un code source plausible pour `Opaque.out`. Ce programme ressemblera à `BufferOverflow.c` mais sera exploitable par `exploit.sh`.

Moralité

Malgré tous les progrès de nos compilateurs,
Quel que soit le langage, attendez-vous au pire.
Ne faites pas confiance, on pourrait bien vous nuire :
Contrôlez les entrées de l'utilisateur.

Domaine public.