Sécurité Logicielle

– Examen (**1**) –

1 Audit de code (8 points)

Pour l'exercice, nous supposerons que le programme suivant est `setuid` et compilé pour `i386` dans toutes les questions qui seront posées. Le but sera d'ouvrir un shell avec les droits du propriétaire de l'exécutable. On omettra le détails des adresses précises, mais on décrira les grandes lignes des attaques, les principes de base ainsi que les pièges éventuels à contourner.

```c
#include <stdio.h>
#include <stdlib.h>

#include <string.h>

void foo (char *name)
{
  char fmt[25] = "Hello %s! How are you?\n";
  char buf[256];

  memset (buf, 0, sizeof (buf));
  strcpy (buf, name);

  printf (fmt, buf);
  puts (buf);
}

int main (int argc, char *argv[])
{
  if (argc < 2)
    return EXIT_FAILURE;

  foo (argv[1]);

  system("/bin/echo 'Boooh!'");

  return EXIT_SUCCESS;
}
```

Questions

1. Imaginons que le programme soit exécuté dans un système sans ASLR, et avec une pile exécutable. Décrivez un moyen simple d'exploiter une des failles du programme en question dans ce contexte.
2. Même question que précédemment mais la pile n'est plus exécutable (NX) cependant l'ASLR reste désactivé. Décrivez une exploitation des failles dans ce contexte.
3. Même question que précédemment mais avecl'ASLR, la pile non-exécutable (NX) et l'ajout de canaries ('`-fstack-protector`') pour protéger la pile.
4. Enfin, comment corrigeriez-vous le programme pour le rendre plus sûr et quelles protections au niveau du système d'exploitation préconiseriez-vous d'utiliser à tout prix ?

2 Escape From Return-Oriented Programming (12 points)

Lisez l'article "*Escape From Return-Oriented Programming : Return-Oriented Programming without Returns (on the x86)*" par S. Checkoway et H. Shacham (Rapport Technique CS2010-0954, UCSD, 2010). Notez que cette technique est aussi souvent appelée le JOP (*Jump-Oriented-Programming*), mais cette dénomination n'est apparue qu'après la publication de l'article.

Questions

1. Rappelez les principes des attaques ROP et dites quelles sont les protections qu'elles permettent de contourner et quelles sont les protections qui rendent leur exploitation plus difficile.
2. Expliquez le principe des instructions "*return-like*" de type '`pop %eax; jmp *%eax`' décrites au début de la section 2.1 et comparez avec des gadgets ROP classiques.
3. De même, expliquez le principe des instructions "*return-like*" de type '`pop %eax; jmp *(%eax)`' décrites à la fin de la section 2.1 et comparez avec des gadgets ROP classiques. Quels avantages (ou inconvénients) ont ces instructions par rapport aux instructions de type '`pop %eax; jmp *%eax`' ?
4. La section 3 explique les spécificités des gadgets que l'on recherche. En vous aidant du texte, expliquez comment vous concevriez l'algorithme de recherche des gadget. Et, expliquez pourquoi les gadgets risquent d'être souvent découverts entre deux instructions valides.
5. Expliquez ce que veut dire "*Turing complet*" dans ce contexte, détaillez le but de l'article et justifiez le choix des auteurs d'avoir pris la `libc` comme exemple de base.
6. Expliquez simplement, les différentes étapes qui permettent de réaliser un "*Data movement*". (voir section 4., "*A Gadget Catalog*")
7. Expliquez les différentes étapes qui composent le "*Function call gadget*" qui est visualisé sur la figure 4 (éventuellement, passez les parties que vous n'avez pas comprises).
8. Les gadgets ROP et JOP peuvent être combinés sans problème, ce qui augmente la surface d'attaque sur les programmes usuels. Listez quelques protections contre les ROP et évaluez les face à des JOP. Éventuellement, suggérez des protections contre les JOP, si vous avez des idées à ce propos.