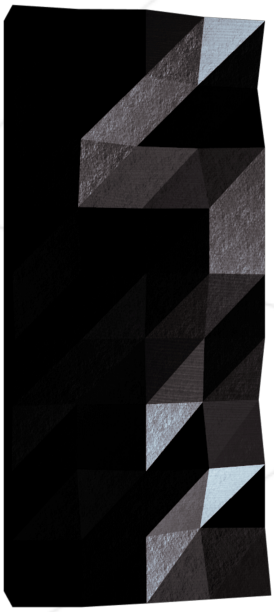

Project UNIX

woody_woodpacker

42 Staff pedago@staff.42.fr

Résumé: Ce projet consiste à coder un simple packer !

Table des matières

Chapitre I

Préambule

Chapitre II

Introduction

Les "Packers" sont des utilitaires dont la tâche consiste à compresser un programme exécutable (.exe, .dll, .ocx ...) et à le chiffrer simultanément. Au moment de son exécution, un programme ainsi passé entre les mains d'un packer est chargé en mémoire compressé et chiffré, puis il se décompresse (et se déchiffre) pour, enfin, s'exécuter.

La création de ce genre de programme est liée au fait que les antivirus analysent généralement un programme au moment de son chargement en mémoire, avant qu'il ne s'exécute. Ainsi, le chiffrement et la compression du packer permettent de contourner simplement ces mesures en obfusquant le contenu de l'exécutable jusqu'à son exécution.

Chapitre III

Objectifs

Le but du projet est de coder un programme qui aura pour tâche, dans un premier temps, de chiffrer un programme passé en paramètre. Seuls les ELF 64 bits seront traités ici.

Un nouveau programme "woody" sera alors généré à la fin de l'exécution du programme. Lorsque ce nouveau programme (woody) sera exécuté, il devra se déchiffrer pour pouvoir se lancer. Son exécution sera identique en tout point avec le premier programme passé en paramétre à l'étape précédente.

Bien que nous n'allons pas voir, dans ce projet, la capacité de compression directement, vous êtes fortement encouragés à explorer les méthodes possibles !

`Le programme, en fonction de l'algorithme choisi, peut être très lent (ou pas vraiment optimisé) dans certains cas : pour pallier à ce soucis, je vous encourage à faire cette partie en assembleur! Le cas échéant, votre Makefile devra contenir les règles de compilation appropriées.`

Chapitre IV

Partie obligatoire

- L'exécutable devra se nommer `woody_woodpacker`.
- Votre programme prend en paramètre un fichier binaire (ELF 64 bits uniquement).
- À la fin de l'exécution de votre programme, un second fichier sera créé, sous le nom de `woody`.
- Vous êtes libres dans le choix d'algorithme de chiffrement sur les binaires.

`La complexité de votre algorithme va néanmoins être un élément important de votre notation. Vous devrez justifier de votre choix en soutenance. Un simple ROT n'est pas considéré comme un algorithme avancé !`

- Dans le cas d'utilisation d'un algorithme basé sur une clé de chiffrements, celle-ci devra être générée de la façon la plus aléatoire possible. Cette clé sera lisible sur la sortie standard au lancement du programme principal.
- Lorsque vous exécutez le programme "chiffré", il devra écrire la string "....WOODY....", suivie d'un retour à la ligne, pour indiquer que le binaire est alors déchiffré. Son exécution, après déchiffrement, ne sera pas modifiée.
- Evidemment, en aucun cas l'exécution du programme "chiffré" ne doit crasher.
- En aucun cas votre programme ne doit modifier le fonctionnement du binaire final créé, son exécution doit correspondre au programme passé en paramètre à `woody_woodpacker`.

- Voici un exemple d'utilisation possible (les binaires sont disponibles dans le fichier `resources.tar`, sur la page projet) :

```
# nl sample.c
 1  #include <stdio.h>
 2  int
 3  main(void) {
 4      printf("Hello, World!\n");
 5      return (0x0);
 6  }
#clang -m32 -o sample sample.c
# ./woody_woodpacker sample
File architecture not suported. x86_64 only
# clang -m64 -o sample sample.c
# ls
sample  sample.c  woody_woodpacker
# ./woody_woodpacker sample
key_value: 07A51FF040D45D5CD
# ls
sample  sample.c  woody  woody_woodpacker
# objdump -D sample | tail -f -n 20
  45:   67 73 2f                addr16 jae 77 <_init-0x80481f9>
  48:   52                      push   %edx
  49:   45                      inc    %ebp
  4a:   4c                      dec    %esp
  4b:   45                      inc    %ebp
  4c:   41                      inc    %ecx
  4d:   53                      push   %ebx
  4e:   45                      inc    %ebp
  4f:   5f                      pop    %edi
  50:   33 36                   xor    (%esi),%esi
  52:   32 2f                   xor    (%edi),%ch
  54:   66 69 6e 61 6c 29       imul   $0x296c,0x61(%esi),%bp
  5a:   20 28                   and    %ch,(%eax)
  5c:   62 61 73                bound  %esp,0x73(%ecx)
  5f:   65 64 20 6f 6e          gs and %ch,%fs:0x6e(%edi)
  64:   20 4c 4c 56             and    %cl,0x56(%esp,%ecx,2)
  68:   4d                      dec    %ebp
  69:   20 33                   and    %dh,(%ebx)
  6b:   2e 36 2e 32 29          cs ss xor %cs:(%ecx),%ch
    ...
# objdump -D woody | tail -f -n 20
 197:   64 69 6e 5f 75 73 65    imul   $0x64657375,%fs:0x5f(%rsi),%ebp
 19e:   64
 19f:   00 5f 5f                add    %bl,0x5f(%rdi)
 1a2:   6c                      insb   (%dx),%es:(%rdi)
 1a3:   69 62 63 5f 63 73 75    imul   $0x7573635f,0x63(%rdx),%esp
 1aa:   5f                      pop    %rdi
 1ab:   69 6e 69 74 00 5f 5f    imul   $0x5f5f0074,0x69(%rsi),%ebp
 1b2:   62 73                   (bad)  {%k7}
 1b4:   73 5f                   jae    215 <(null)-0x400163>
 1b6:   73 74                   jae    22c <(null)-0x40014c>
 1b8:   61                      (bad)
 1b9:   72 74                   jb     22f <(null)-0x400149>
 1bb:   00 6d 61                add    %ch,0x61(%rbp)
 1be:   69 6e 00 5f 5f 54 4d    imul   $0x4d545f5f,0x0(%rsi),%ebp
 1c5:   43 5f                   rex.XB pop %r15
 1c7:   45                      rex.RB
 1c8:   4e                      rex.WRX
 1c9:   44 5f                   rex.R pop %rdi
 1cb:   5f                      pop    %rdi
    ...
# ./sample
Hello, World!
# ./woody
....WOODY.....
Hello, World!
```

Chapitre V

Partie bonus

Les bonus ne seront comptabilisés que si votre partie obligatoire est PARFAITE. Par PARFAITE, on entend bien évidemment qu'elle est entièrement réalisée, et qu'il n'est pas possible de mettre son comportement en défaut, même en cas d'erreur aussi vicieuse soit-elle, de mauvaise utilisation, etc ... Concrètement, cela signifie que si votre partie obligatoire n'est pas validée, vos bonus seront intégralement IGNORÉS.

Des idées de bonus :

- Support 32bits .
- Utilisation de clé paramétrable.
- Optimisation de l'algorithme utilisé via de l'assembleur.
- Support de différents formats de binaire (PE, Mach-O..)
- Compression du binaire.

Chapitre VI

Rendu et peer-évaluation

- Ce projet ne sera corrigé que par des humains. Vous êtes donc libres d'organiser et nommer vos fichiers comme vous le désirez, en respectant néanmoins les contraintes listées ici.

- Vous devez coder en C (la version n'est pas imposée ici) et rendre un Makefile (respectant les règles habituelles).

- Dans le cadre de votre partie obligatoire, vous avez le droit d'utiliser les fonctions suivantes :
  - open, close, exit
  - fpusts, fflush, lseek
  - mmap, munmap
  - perror, strerror
  - syscall
  - les fonctions de la famille printf.
  - les fonctions autorisées dans le cadre de votre libft(read, write, malloc, free, par exemple :-)).
  - Vous avez l'autorisation d'utiliser d'autres fonctions dans le cadre de vos bonus, à condition que leur utilisation soit dûment justifiée lors de votre correction. Soyez malins.

- Vous pouvez poser vos questions sur le forum, sur jabber, IRC, slack...