UNIX Project

woody_woodpacker

42 Staff pedago@staff.42.fr

Summary: This project is about coding a simple packer!

Contents

Chapter I

Foreword

Chapter II

Introduction

“Packers” are tools that consist on compressing executable programs (.exe, .dll, .ocx ...) and encrypt them simultaneously. During execution, a program passing through a packer is loaded in memory, compressed and encrypted, then it will be decompressed (decrypted as well) and finally be executed.

The creation of this kind of program is linked to the fact that antivirus programs generally analyse programs when they are loaded in memory, before they are executed. Thus, encryption and compression of a packer allow to bypass this behavior by obfuscating the content of an executable until it execution.

Chapter III

Objectives

The goal of this project is to code a program that will, firstly, encrypt a program give as parameter. Only 64 bits ELF files will be managed here.

A new program called “woody” will be generated from this execution. When this new program (woody) will be executed, it will have to be decrypted to be run. Its execution has to be at any point identical to the program given as parameter in the last step.

Even though we will no see, in this project, the compression possibilities directly, we strongly advise you to explore the possible methods!

`The program, depending on the chosen algorithm, can be really slow (or not really optimized) in some cases: to counter this problem, I advise you to do this part in assembly language! If you do your makefile will require the appropriate compilation rules.`

Chapter IV

General Instructions

- This project will be corrected by humans only. You're allowed to organise and name your files as you see fit, but you must follow the following rules.
- Your project must be written in C (the version is up to you) and submit a makefile with the usual rules.
- Within the mandatory part, you are allowed to use the following functions:
  - open, close, exit
  - fpusts, fflush, lseek
  - mmap, munmap
  - perror, strerror
  - syscall
  - the functions of the printf family
  - the function authorized within your libft (read, write, malloc, free, for exemple :-)).
- You are allowed to use other functions to complete the bonus part as long as their use is justified during your defence. Be smart!
- You can ask your questions on the forum, on slack...

Chapter V

Mandatory part

- The executable must be named `woody_woodpacker`.
- Your program takes a binary file parameter (64 bits ELF only).
- At the end of the execution of your program, a second file will be created named `woody`.
- You are free to choose the encryption algorithm.

```
The complexity of your algorithm will nonetheless be a very important
part of the grading.  You have to justify your choice during p2p.  A
simple ROT isn't considered an advanced algorithm !
```

- In the case of an algorithm based on an encryption key, it will have to be generated the most randomly possible. It will be displayed on the standard output when running the main program.
- When running the "encrypted" program, it will have to display the string "....WOODY....", followed by a newline, to indicate that the binary is encrypted. Its execution after decryption will not be altered.
- Obviously, in no way the "encrypted" program is allowed to crash.
- You program musn't modify the running of the final binary produced, its execution must correspond the binary give as parameter to `woody_woodpacker`.

- Here is a possible example, (binaries are available in the `resources.tar` file, on the project's page:

```
# 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!
```

Chapter VI

Bonus part

We will look at your bonus part if and only if your mandatory part is EXCELLENT. This means that your must complete the mandatory part, beginning to end, and your error management needs to be flawless, even in cases of twisted or bad usage. If that's not the case, your bonuses will be totally IGNORED.

Find below a few ideas of interesting bonuses:

- 32bits management.
- Parameterized key.
- Optimisation of the used algorithm through assembly language.
- Additional binary management (PE, Mach-O..)
- Binary compression.

Chapter VII

Submission and peer-evaluation

Submit your work on your GiT repository as usual. Only the work on your repository will be graded.