Contents

Hénallux
Section sécurité des systèmes

Développement

Implémentation d'un IDS

Projet réalisé par

Mustafa-Can KUS
Jordan DALCQ

Année académique 2020-2021

Contents

Part I

Introduction

Chapter 1

Contexte

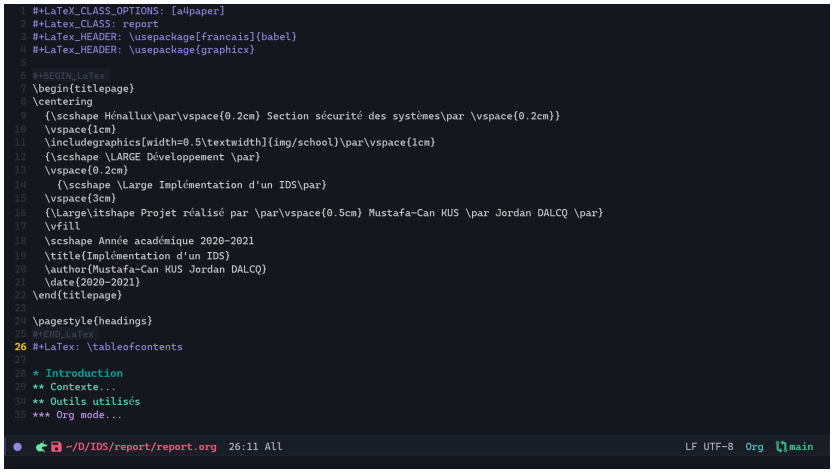
Dans le cadre du cours de développement il nous a été demandé de réalisé un système de détection d'intrusion (ou IDS pour faire plus cours). Ce programme à pour but d'analyser le trafique réseau et reporter toutes activité suspectes à un administrateur système ou même un administrateur réseau grâce à un système de log.

Chapter 2

Outils utilisés

2.1 Org mode

Org mode est un mode majeur pour le logiciel GNU Emacs qui permet de prendre notes et maintenir une Todo liste et permet aussi de planifier facilement des projets grâce à son langage Markup (très proche du markdown). On l'a utilisé pour écrire ce report (ok on a un peu tricher on a mit un peu de Latex pour faire joli) et aussi pour planifier notre travail



2.2 Github

Une fois notre planing fait il nous fallait une solution pour que chaqu'un d'entre nous aie une copie du code toujours à jours et qu'on puisse tracker nos modifications, ce qui est très pratique en cas de bug, en effet il nous aurait suffit que de revenir quelques modifications en arrière et le problème est régler !

Notre projet est disponible ici: #+BEGIN_LaTex

`https://github.com/Les-IRaniens/IDS`

#+END_LaTex

2.3 GNU Makefile

Comme tous bon informaticiens qui se respecte, on a pas envie de tapper une très longue commande composé d'une dixène de fichiers et d'une autres dixène de flag à chaque fois qu'on souhaite compiler notre programme, alors on a décider d'utiliser un makefile. Le makefile s'occupe de compiler et de linker notre code automatiquement, il suffit de tapper make dans le terminal et le tour est jouer !

2.4 Nos flags de compilations

- -pednatic: nous oblige fortement à adhérer aux règles de l'ANSI C
- -Wpednatic: nous affiche des warnings si on respecte pas la pedantic
- -Wall: nous permet d'avoir tous les warnings sur des pratiques considérées comme questionnable
- -Wextra: Couvre encore plus de warnings que -Wall
- -Werror: Transforme tous les warnings en erreur (Oui on est sans pitier ici)
- -g: Permet d'avoir les symboles de debugger
- -Isrc/: Permet d'inclure facilement les headers du dossier src
- -fsanitize=undefined & -fsanitize=undefined: Permet de tracker chaque memory leaks et nous dit sur quelle ligne est le problème
- -lpcap: Inclus la libpcap à notre projet

2.5 ArchLinux

Programmer sur Kali c'est pas top, surtout sur une VM ! Donc on a préférer utiliser ArchLinux pour le travail sur machine native. Pourquoi celà ? Car Archlinux est une distribution polyvalante qui a TOUS les paquets qu'on désire (oui même tous les paquets de Kali)

2.6 Vscodium

C'est Visual Studio Code - les fonctions de télémétries, c'est notre éditeur de choix, car il permet une super bonne intégration avec notre github, nous informe de nos erreurs et des warnings potentiels grâce à l'extension C/C++.

Part II

Explications du code

Dés le départ nous avons penser à subdiviser le code en plusieurs fichier, celà permet de limiter le nombre de ligne de code par fichier, en effet le maximum par fichier est de 300 lignes de code, se qui permet de trouver nos erreurs plus efficacement

Chapter 3

main.c

3.1 int main (int argc, char *argv[])

C'est un peux le chef d'orchestre de notre programme, ce fichier contient la fonction main qui donne les entrées utilisateurs au différentes fonctions grâce aux paramètres donné au programme (argv) et vérifie que l'utilisateur démarre bien le programme avec \.

Chapter 4

usage.c

4.1 void show\usage (char *exe\path)

Si l'utilisateur nous donne aucun paramètre ou si il nous donne le paramètre d'aide (-h | –help) la fonction main nous envois sur cette fonction et nous montre un message d'utilisation. L'argument exe\path comprend le chemain de l'executable et on récupère que le nom de l'executable (grâce à la fonction basename)

Chapter 5

checkuser.c

5.1 bool is\root(void)

il s'agit d'une simple vérification pour voir si on démarre bien le programme en tant qu'administrateur système

Chapter 6

utils/list.c

6.1 void init\str\list(StrList *self)

Cette fonction est le constructeur de notre objet StrList (il s'agit d'une structure qui correspont à un tableau de string allouer de manière automatique) Il attribue un emplacement mémoire de base.

6.2 void append\str\list(StrList *self, const char *str)

La fonction vérifie tous d'abord si notre liste à assez de place en mémoire, si c'est pas le cas on lui donne plus d'emplacement mémoire et on ajoute la chaine de carractère donner en paramètre à la fin de la liste

6.3 StrList split\to\liststr(char *str, const char *del)

Cette méthode découpe une chaine de carractère selon un délimiteur et met chaque partie dans une liste

6.4 void free\str\list(StrList *self)

Cette fonction libère la mémoire occupé par une liste

Chapter 7

utils/log.c

7.1 void log_ids(const char *msg)

Cette fonction écrit dans le syslog

Chapter 8

readrules.c

8.1 StrList read_rules(const char *filename)

C'est ici que se produit la lecture des règles (données en paramètre par l'utilisateur), le fichier est lu ligne par ligne et placer dans une liste de string

Chapter 9

rule.c

9.1 RuleList parse_rule(StrList rules)

Voilà la fonction qui est responsable de traduire chaque règles en une structure qu'on a simplement nommer \, qu'on place dans une liste qu'on a nommé \ qui occupe un certains emplacement mémoire; la struct rule contient le protocol, l'addresse de source et de destination, le prot de source et de destination.

9.2 bool is_in_context(Rule rule, void *packet, Protocole proto)

Cette fonction vérifie si le paquet match avec une des règles

9.3 free_rules(RuleList *lst)

cette fonction libère l'emplacement mémoire pris par les listes de règles