RAPPORT
Projet Sécurité

Table des matières

Bibliothèque GTK :

Pour l'ensemble de ce projet, nous allons utiliser la bibliothèque GTK afin de créer des interfaces graphique pour nos programmes exécutables. Nous avons fait ce choix car cette bibliothèque possède une documentation avancée et est plus facile d'implémenter que d'autre solution graphique.

Pour créer une interface graphique simple à l'aide de GTK,

on initialise premièrement l'application avec : "gtk_init(&argc, &argv)". Pour construire notre fenêtre principale, on utilise : "gtk_window_new()", on peut également mettre le paramètre "GTK_WINDOW_TOPLEVEL" afin que cette dernière soit indépendante. Suite à cela, on peut initialiser la taille et le titre de notre application grâce aux fonctions : "gtk_window_set_title" et "gtk_window_set_default_size".

```
GtkWidget *window = gtk_application_window_new(app);
gtk_window_set_title(GTK_WINDOW(window), "Démo GTK");
gtk_window_set_default_size(GTK_WINDOW(window), 400, 300);
```

Pour rajouter un bouton dans notre interface, on créer un nouvel objet GtkWidget que l'on initialise ensuite grâce à "gtk_button_new_with_label("Text du bouton")". Afin de connecter une action au clic de notre bouton, on peut utiliser la fonction "g_signal_connect". Notre bouton n'est pour l'instant pas visible dans notre interface graphique, pour remédier à cela, il nous faut le rajouter dans la fenêtre grâce à "gtk_container_add".

```
GtkWidget *button = gtk_button_new_with_label("Cliquez ici");
gtk_box_pack_start(GTK_BOX(box), button, FALSE, FALSE, 0);
g_signal_connect(button, "clicked", G_CALLBACK(button_clicked), label);
```

Pour ajouter un Text, les mêmes étapes sont requises : créer un objet GtkWidget et l'initialiser cette fois ci avec "gtk_label_new(("Text")". Puis on l'ajoute à la fenêtre avec "gtk_container_add".

```
GtkWidget *label = gtk_label_new("Bonjour, ceci est un texte !");
gtk_box_pack_start(GTK_BOX(box), label, FALSE, FALSE, 0);
```

Pour ajouter une image, on créer une nouvelle fois un Widget

```
GtkWidget *image = gtk_image_new_from_file("test.png");
gtk_box_pack_start(GTK_BOX(box), image, TRUE, TRUE, 0);
```

Une fois tout les éléments de notre application créés, on affiche la fenêtre et tout les widgets (”gtk_widget_show_all”) et on lance une boucle spéciale, qui va garder la fenêtre ouverte durant tout le temps d’utilisation (”gtk_main()”)

```
gtk_widget_show_all(window);
```

Récupérer les images

La première partie de notre travail a été de récupérer les images pour pouvoir les utiliser. C’était alors l’occasion de revenir dans le développement en C en utilisant des listes faites sous forme de pointeurs de chaînes de caractères.

Nous avons alors mis en place plusieurs fonctions :

- verifier_extension()
- charger_images()
- taille_liste()

Vérifier l’extension

Il était assez simple de vérifier l’extension des différents dossiers. Nous prenons une chaîne de caractères de 6 octets, dans laquelle nous mettons ce qui se trouve après le point si l’on en trouve un dans le nom du fichier. On le compare avec les différentes extensions acceptées: {".jpg", ".png", ".bmp", ".jpeg", NULL}si on trouve une de ces chaînes, on renvoi alors True (1).

Taille de la liste

Permet simplement de vérifier la taille d’une liste en la parcourant et en incrémentant une valeur.

Charger l’image

La fonction charger_image() est la plus complexe, dans celle-ci, nous cherchons premièrement les différents fichiers présents dans notre répertoire courant, pour premièrement faire la taille de tous les éléments que nous devons prendre, bien entendu, seulement si leur extension est vérifiée. On alloue alors une chaîne de caractères spécialement à la taille du nombre de fichiers qui ont l’extension qui est vérifiée. Et tous ces éléments sont revérifiés et passer dans une liste de chaîne de caractères.

Table

Nous avons décidé de faire cette composition avec un *gtk_table_new* qui nous permet simplement de mettre en place les différents éléments.

Mise en place de l’image

Les données que nous allons utilisé ont été mises dans une structure: AppData permettant de gérer les images, elle contient :

- Le widget de l’image
- toutes les images (retour de charger_image())
- le nombre d’images au total
- l’indice actuel de l’image
- le widget redimensionné pour qu’il n’y ait pas de problème
- la boite qui permet un meilleur affichage des éléments

Les boutons

Chaque bouton est relié à une fonction, qui décrémente ou incrémente l’index en fonction du bouton passé et qui change l’image avec une nouvelle fonction.

Celle-ci utilse la fonction modifier_image qui permet de changer l’image en reprenant l’index.

Cette fonction appelle une dernière fonction que nous avons mise en place redimensionner_image qui nous permet de ne pas avoir des images qui ne sont pas à la bonne taille et qui dépasse de ce que l’on veut. Celle-ci prend en compte la taille totale de la boite dans laquelle nous avons l’image, et en ressort une taille convenable. C’est l’un des plus gros problèmes que nous avons rencontré lors de notre mise en place de ce MediaPlayer.

Nous pouvons voir notamment que dans les fonctions, nous utilisons des gboolean et des gpointer, plus adapter à l’utilisation de GTK. Une fois que l’index est changé et que l’image est redimensionnée on voit alors que l’image change bel et bien.

Virus

La plupart des informations que nous allons donner ici sont guidées dans le polycopié.

Nous allons donc vous l’expliquer l’utilisation du virus en suivant l’exécution du code :

Lancement du code

Dans le main, partie virus se lance seulement avec une seule ligne:

infecter_fichiers(argv[0]);

Cette ligne permet de démarrer le processus du virus compagnon en donnant comme paramètre *argv[0]* qui correspond au nom du programme lancé par le script.

Recherche des fichiers cibles:

La fonction *infecter_fichiers(const char chemin) permet d'infecter tous les fichiers du répertoire. Ceux-ci sont alors pris avec, comme dis dans le polycopié, l'utilisation d'opendir() dans lequel une boucle for passe pour récupérer les noms des fichiers.

On teste alors plusieurs choses sur ces fichiers, premièrement qu'il ne correspond pas à lui-même, ce qui pourrait causer des problème, mais nous avons aussi d'autres vérifications :

Est ce que le fichiers est exécutable et valide ?

On appelle pour cela la fonction *est_executable(const char fichier) qui va simplement, avec stat, on teste d'abord si le fichier existe bien:

if (stat(fichier, &st) == -1)

Puis enfin, on regarde s'il correspond bien à un exécutable que l'utilisateur actuel peut exécuter:

return S_ISREG(st.st_mode) && (access(fichier, X_OK) == 0)

S_ISREG(st.st_mode) cette partie vérifie que le fichier est régulier, (access(fichier, X_OK) == 0) vérifie que le fichier est bien exécutable, mais surtout par l'utilisateur.

Est ce que le programme est déjà infecté:

Nous faisons deux vérifications:

- Si le fichier est déjà un .old: *if (strstr(fichier, ".old") != NULL)*
- S'il existe déjà un .old de ce fichier dans le répertoire:
    - Création d'une chaine de caractères du nom du .old:*snprintf(old, sizeof(old), "%s.old", fichier);*
    - Test de voir si ce fichier se lance bien: *FILE *f = fopen(old, "r"); if(f) {fclose(f); return 1;}*
- Si tout est vérifié, on retourne 0 est le fichier n'est donc pas infecté.

Exécution de notre virus

Maintenant, que nous avons bien vérifié que le fichier que nous avons dans notre répertoire est bien exécutable et n'est pas infecté, nous pouvons passer à la mise en place du virus dans ce fichier.

Nous appelons la fonction :*infecter(const char *cible, const char chemin).

La cible correspond à notre fichier cible et le chemin est simplement le chemin de notre fichier actuel qui exécute le virus.

Cette fonction est un peu plus longue, pour vous la détailler, nous devons d'abord vous expliquer ce qu'est un "retour en arrière". En cas de problème dans la fonction infecter(…), nous avons mis en place des retours en arrière:

- Nous testons d'abord de renommer le script cible avec un ".old" ajouté

    - Si la modification ne marche pas nous renvoyons direct la fonction
- Nous essayons d'ouvrir le fichier source, celui sur lequel on lance le script
    - Si cette ouverture ne marche pas:
        - Retour en arrière: on re modifie le nom
        - On renvoie une valeur pour dire que ce n'a pas marché
- Enfin, on crée un nouveau fichier du même nom que notre cible qui va porter le virus
    - Si celui-ci ne se crée pas: retour en arrière et renvoie d'une valeur de base.

Ces différents retours en arrière permettent de ne pas avoir un script qui, en cas d'erreur, met un désordre dans les fichiers.

Le reste du code est donc assez simple:

- On réécrit une par une les octets de notre exécutable source dans l'exécutable de destination.
    - Si, pour une certaines raison, l'écriture ne marche pas: on fait un retour en arrière et on retourne une valeur par défaut.

Nous finissons par un printf(), pour débugger et que ce script soit plus parlant lorsqu'il est lancé. Bien entendu, ce genre de print ne se fait pas dans un virus réel, ceux-ci sont présents à but de comprendre le code, vérifier et prouver que celui-ci fonctionne.

Transférer l'exécution

En effet, le virus n'est pas tout à fait fini. Dans l'état actuel des choses, si nous lançons notre virus MediaPlayer et que nous relançons un autre script, par exemple ./pi nous pourrons nous apercevoir que celui-ci lancera également le MediaPlayer. Comment gérer ce problème ?

Nous avons alors créé une fonction permettant de transférer l'exécution de notre code: *transferer_execution(int argc, char argv[])

Cette fonction, assez simple, permet de savoir premièrement si le script que nous lançons est le MediaPlayer ou non: *char nom = basename(argv[0]);if (strcmp(nom, "MediaPlayer") != 0);

Nous prenons dans les paramètres, bien entendu, les arguments passés lorsque le script est lancé, premièrement pour savoir le nom du script lancé et le comparer avec "MediaPlayer" pour savoir si notre code doit lancé la suite du MediaPlayer ou:

- Prendre le fichier « .old »: *snprintf(old, sizeof(old), "%s.old", nom);*
- Vérifier que nous avons bien l'accès à l'exécution: *if (access(old, F_OK) == 0)*
- Le lancer avec les arguments pris auparavant: *execv(old, argv);*
- Si nous avons une erreur dans l'exécution, on revoit un *perror()* et on arrête le script.

Nous pouvons voir qu'execv va arrêter le script en cours, comme l'exit(0) et le MediaPlayer ne va alors pas se lancer.

Cette fonction est présente dans la fonction main, juste après l'exécution de notre virus.

Questions

Dans le cadre général d'un virus, préciser quelles sont les grandes fonctions que met en œuvre un virus lors d'une attaque (mécanisme d'attache) et leurs enchaînements ?

Les grandes fonctions sont : l'infiltration dans le système cible, la dissimulation derrière un programme simple (ici le MediaPlayer), la recherche de cibles (Ici les exécutables du répertoire, la propagation (remplacement des cibles), la charge virale, c'est-à-dire exécuter l'action malveillante (que l'on n'a pas ici) et pour finir le transfert d'exécution.

Pourquoi, d'après vous, dans le cas du virus compagnon, cette vérification [de sur-infection] doit-elle être double ?

Premièrement, si nous essayons d'exécuter notre virus sur un fichier ".old" et deuxièmement si nous essayons d'exécuter notre virus sur un fichier qui a déjà son ".old".

D'après vous, que se produit-il si l'étape 3 (rendre la copie exécutable) est oubliée ou ne fonctionne pas ?

Le fichier ne sera pas exécutable et donc la victime détectera rapidement le virus.

D'après vous, comment, à travers cette étape (6) (infection des fichiers cibles), peut-on amplifier l'infection ?

Nous avons réduit les fichiers exécutables cibles à ceux du répertoire courant, si nous prenons, par exemple, le répertoire père et les répertoires fils, nous pourrions infecter rapidement une grande partie de la machine.

D'après-vous, que se produit-il si l'étape 7 (transfert d'exécution au code hôte) est oubliée ou ne fonctionne pas ?

Les autres programmes vont alors tous lancer le MediaPlayer à la place de leur .old et donc la discrétion du virus est corrompu.

Tests du virus

Au lieu de faire plusieurs captures d'écran, nous avons préféré montrer directement ce que fait notre virus et comment il se lance :

https://youtu.be/h1y1_L1O6tE?si=4D5BUFq_iydxxMe0

Exécutables :

Cryptage d'un string en code de César :

Le code de César est un moyen de cryptage simple qui consiste à modifier la place des lettres dans l'alphabet du message entré par rapport à une clé elle aussi entrée.

Pour cela nous allons récupérer, via la fonction “scanf”, le message et la clé de décalage que l’utilisateur aura entré dans le terminal.

Suite à cela on calcule le code en décalant chaque lettre du mot en fonction de la clé entrée, en faisant attention à revenir au début de l’alphabet si jamais le décalage va au-delà de 26. Enfin on affiche le résultat dans le terminal

Exécutable :

```
tpuser@m-ulr-u22-r192n:~/Téléchargements/ProjetSecu$ ./cesar
Entrez votre message : test
Entrez la clé de décalage : 3
Message chiffré : whvw
```

Deviner le nombre :

Ce programme est un petit jeu dont le but est de trouver un nombre aléatoire entre 1 et 100 que le programme choisira lors de son lancement, le tout avec le moins de tentatives possible. Afin d’aider le joueur, le programme indiquera si le nombre que l’on a entré est plus grand ou plus petit que celui que l’on doit deviner.

Pour commencer, on va créer une structure qui va contenir le nombre que l’on doit deviner ainsi que le nombre de tentative effectué. On appellera cette structure GameData. Ensuite dans une fonction “play_game” on va initialiser un nouveau nombre à deviner et mettre à 0 le nombre de tentatives avec les lignes :

Puis on va bouclé tant que le joueur n’aura pas trouver le nombre. Dans cette boucle on va venir, à l’aide d’un scanf, récupérer la supposition du joueur sur le nombre à deviner. On va donc comparer cette supposition au nombre à trouver. Suivant si cette dernière est supérieur ou inférieur à nos nombre, le programme renverra une indication au joueur. Une fois le nombre trouvé, le joueur pourra décider de rejouer ou non grâce aux lignes :

```
tpuser@m-ulr-u22-r192n:~/Téléchargements/ProjetSecu$ ./jeu
Devine un nombre entre 1 et 100 !
Entre ton nombre : 50
Trop haut ! Tentatives : 1
Entre ton nombre : 25
Trop haut ! Tentatives : 2
Entre ton nombre : 15
Trop haut ! Tentatives : 3
Entre ton nombre : 10
Trop haut ! Tentatives : 4
Entre ton nombre : 5
Trop haut ! Tentatives : 5
Entre ton nombre : 1
Trop bas ! Tentatives : 6
Entre ton nombre : 3
Trop haut ! Tentatives : 7
Entre ton nombre : 2
Bravo ! Trouvé en 8 tentatives !
Veux-tu rejouer ? (o/n) :
```

Calcule des racines d’un polynôme du 2nd degré :

Le calcul des racines d’une équation du type : ax^2+bx+c se résout en calculant delta ($b^2 - 4ac$) et en calculant ensuite les racines en fonction du signe de delta. Pour cela rien de plus simple en C. On récupère les valeurs de a, b et c entré par l’utilisateur puis on va appliqué les calculs de delta et des racines.

Pour ce programme, nous avons implémenter une interface graphique avec GTK afin d’entrer les valeurs de coefficients et d’afficher les réponses sous forme d’un graphique avec la courbe représentative de la fonction créer avec les entrées.

Pour simplifier les calculs, on utilise une structure “Polynome” qui contient les coefficient a,b et c, les potentiels racines r1 et r2 et le nombre de racines qui sera déterminé par le signe de delta. On utilsie également une autre structure afin de stocker les 3 entrées de l’utilisateur dans le but de simplifier le code. Cette dernière possédera 3 GtkEntry.

La fonction “activate” permet d’initialiser tout les widgets ainsi que la fenêtre de notre interface graphique.

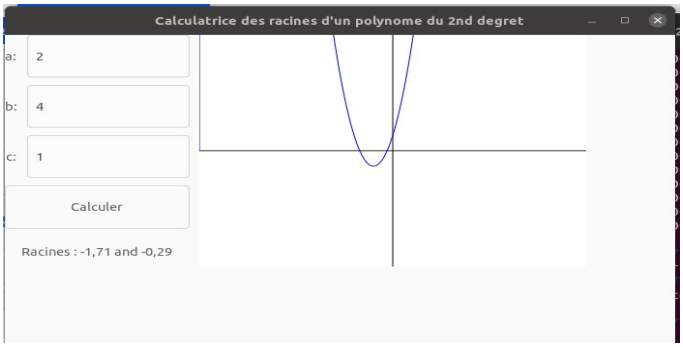
La fonction “on_calculate_clicked” va en premier lieu récupérer les entrées et les stocker dans des variables char grâce à : “gtk_entry_get_text”. Elle va ensuite convertir les chaines de charactères récupérer en entier avec la fonction “atoi” pour ensuite appeler “calculRacines”. Enfin pour afficher le résultat, on va vérifier combien de racines possède le polynôme que l’on vient de calculer en on affiche les résultats en conséquence.

La fonction “affiche_courbe” va permettre de dessiner la courbe représentative du polynôme final avec les coefficients entrés par l’utilisateur. Pour cela, on utilise cairo, une bibliothèque souvent utilisé avec GTK afin d’afficher des dessins et notamment des courbes. On commence alors par créer notre plan 2d avec les axe des abscisses et des ordonnés :

On créer ensuite la courbe de notre fonction d’une autre couleur en bouclant sur les pixel des abscisses (x). Pour chaque pixel, on va alors calculer les coordonnées y avec :

En résumé, on calcule ici simplement en mode brutal toute les valeurs de y pour x. On trace ensuite une ligne point par point avec :

Et on dessine la courbe une fois terminé avec “cairo_strocke”.



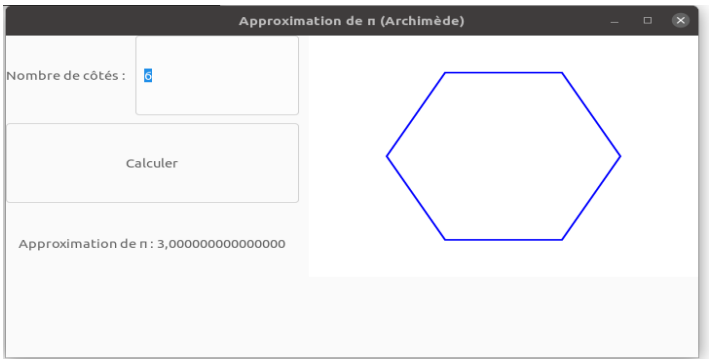
Calcule des décimal de pi avec la méthode d’Archimède :

La méthode d’Archimède, découvert vers 250 avant J-C par le célébré scientifique du même nom, permet d’obtenir une approximation de π en calculant le périmètre d’un polynôme régulier. C’est à dire que plus le polynôme aura de côtés, plus il se rapprochera d’une forme sphérique et donc, plus le calcul de son périmètre se rapprochera de la valeur de π.

La fonction qui permet de calculer cette approximation n’est pas une méthode complexe, elle prend simplement en paramètre un entier n (le nombre de côté du polynôme) :

Pour la partie graphique de ce programme, on créer une fonction “activate” qui va s’occuper de l’ensemble des initialisations graphiques que l’on aura besoin.

On créer alors une fenêtre, une grille pour ordonner nos éléments, plusieurs texte dont un pour entré le nombre de côtés du polynôme, un bouton pour dessiner la figure et calculer l'approximation et une zone de dessin où la figure sera dessiner.



Une fois le bouton cliqué, on va appeler la fonction pour calculer l'approximation de π, puis on va appeler la fonction qui va nous permettre de dessiner la figure correspondant au nombre de côtés entrée par l'utilisateur. Pour cela, on va utiliser la bibliothèque "cairo" que l'on à déja utiliser pour dessiner le graphe des polynôme du 2nd degrés. Une fois les coordonnés du centre de notre figure calculer, on va recalculer les sommets de notre figures en fonctions des paramètres que l'on aura passé dans la fonction. On utilise ensuite "cairo_move_to" pour dessiner point par point la figure.

Simulation de gravité :

Ce programme créer une simulation de gravité en calculer en temps réel l'ensemble des collisions possibles, de façon à recréer un environnement physique réel. L'utilisateur peut, à l'aide d'un clique gauche dans la fenêtre, faire apparaitre une balle bleu qui sera influencer par la gravité. Les balles peuvent entrées en collision avec d'autre balles et avec les bordures de la fenêtre, ce qui d'augmenté le réalisme de la simulation.

On commence par définir les constances qui vont nous permettre de définir notre environnement. La gravité, la friction (réduction de la vitesse en fonction du temps) et le rayon des balles.

On va ensuite créer 2 structures. Une première qui nous fera référence à un objet balle, qui possède donc des coordonnée (x,y) et une vitesse sur 2 axes (vx, vy). La deuxième structure permettra de stocker l'ensemble des balles instanciées par l'utilisateur.

A chaque clique gauche effectué dans la fenêtre, on appellera la fonction add_ball qui ajoutera une balle dans notre fenêtre. Cette fonction créer juste un nouvelle objet Ball grâce à la structure que l'on a définis plus tôt et assimile ses coordonnées à celle de la souris lors du clique et ses vitesses à 0, sans oublier de l'ajouter à la liste des balles.

Pour dessiner ces balles, on utilise une nouvelle fois la bibliothèque 'cairo' et sa fonction 'cairo_arc' qui nous permet de dessiner des cercles.

Dans la fonction main on va ajouter un 'g_timeout_add(x, update, drawing_area)' qui va nous permettre de faire une mise a jour de l'interface tout les x temps en rappelant la fonction 'update'.

Cette fonction update sera donc appelée un nombre de fois considérable. Cette fonction contient elle même un appel vers une fonction de gtk qui permet de redessiner la fenêtre et vers une autre fonction qui va permettre de calculer toute la partie physique de notre programme.

La fonction ‘update_physique’ s’occupe donc de configurer la physique de notre programme à chaque appel de la fonction update.

Pour ajouter un semblant de physique à nos balles, on va commencer par ajouter notre constante de gravité à la vitesse sur l’axe y (vy), on met ensuite à jour la vitesse et on réduit la vitesse grâce a la constance de friction pour créer une simulation de perte d’énergie et de vitesse et cela, pour chaque balles dans la liste des balles instanciées.

Suite à cela, on calcule les collision que les balles peuvent avoir. Le premier type de collision qu’un balle peut avoir est contre les bords de la fenêtre. Pour calculer cela, rien de plus simple, si la balle tape contre le bord du bas (par exemple), donc que sa position en X moins le diamètre constant d’une balle soit inférieur à 0, alors on va faire correspondre sa position en y à son diamètre et on va inverser sa vitesse de sorte à créer un rebond.

On fait ensuite de même pour les 3 autres bords de la fenêtre en modifiant les différents paramètres de positions, de vitesses et de calcule de la condition de collision.

Mais une balle peut aussi entrée en collision avec une autre balle. Pour calculer cela, on va créer une double boucle entre toute les balles contenus dans notre liste de balles instanciées. Puis, pour chaque paire de balles, on va vérifier si l’une n’est pas entrée en collision avec l’autre.

Si cette condition est vrai on va calculer la force produit par la collision pour ensuite modifier leur position et appliquer des vitesses inverses afin de créer un rebond entre les 2 balles concernées.

Pour calculer ce rebond, on va calculer le vecteur de collision n en divisant la distance entre les centre avec la distance entre les 2 balles. On calcule ensuite la vitesse relative en soustrayant la vitesse de la balle2 à celle de la balle1. Puis on calcule le produit scalaire entre cette vitesse relative et le vecteur de collision. Enfin on va calculer la force d’impulsion puis on va ajouter aux vitesses des deux balles, le produit de cette nouvelle valeur d’impulsion avec les valeur x ou y du vecteur de collision.

Pour terminé, on veut éviter que les balles se chevauchent. Pour cela, on va calculer la distance de pénétration (overlap), ce qui veut dire à quelle point l’un est entré dans l’autre. On ajoute alors aux coordonnées overlap (ou -overlap) multiplier par le vecteur de collision que l’on divise par 2 pour éviter d’avoir une force d’impact trop importante.