

JOIN

THE HACKADEMY SCHOOL

Cours de sécurité informatique et de
hacking éthique pour pros et débutants

Dans toute la France et dans plusieurs villes d'Europe

- Initiation Internet/sécurité
- Hacking Newbie, hacking Pro, wifi, architecture sécurisée
- Initiation HTML / PHP / Perl / C
- Initiation Linux / OpenOffice

01 40 21 01 20

DEMANDEZ MÉDÉRIC

EDITO

Habitants de la planète Windows, autochtones de l'univers Unix, ce numéro dédié à la programmation en C a été conçu pour un public hétérogène. Cela fait un an que vous essayez de vous mettre au C sans jamais avoir trop compris ni réussi ? Ou peut-être avez-vous saisi l'esprit du C mais aimeriez en découvrir davantage ?

Sans être un guide complet pour apprendre le langage C de A à Z, ce numéro est un magazine de chevet ayant pour vocation de vous initier, tout d'abord, aux subtilités natives de ce langage qui traverse les âges. Découvrez donc les méthodes avancées de travail sur les pointeurs (algorithmes, manipulations), abordez des articles essentiellement orientés pratique pour la création d'outils, accompagnés parfois d'exercices et de conseils méthodologiques afin de vous permettre d'assimiler plus rapidement le dense contenu que nous avons concocté.

Création d'un contrôleur d'intégrité, initiation à l'OpenGL, développement d'arbres binaires... Les sujets abordés sont aussi divers que peuvent l'être vos centres d'intérêt. Un article d'exploitation avancée originale sur les buffers overflow constitue la petite surprise que nous a réservée l'équipe de n0name, mais il ne s'agit là que d'une goutte d'eau dans cet océan d'informations.

En espérant qu'elle sera la plus enrichissante possible, je vous souhaite, amis lecteurs, une excellente lecture.

Clad Strife

Post Scriptum : gros greet à Crashtfr, ce rôleur qui mérite bien que je le salue ici :-)

sommaire

La chaîne de compilation	p5
Règles de programmation	p9
Écrire un Makefile	p14
Les pointeurs	p18
Les tableaux de pointeurs	p20
Les structures	p23
Les listes chaînées	p26
Récursivité	p29
Assembleur inline	p30
Gdb	p32
Arbres binaires	p36
Coder un contrôleur d'intégrité	p41
Gestion du clavier	p50
Ret into ret	p54
Les bases d'OpenGL	p58
Return into libc	p63
Coder un sniffer TCP	p65
Forger des paquets sous Linux	p71
Initiation à ncurses	p76

THE HACKADEMY PROG

est une publication de DMP

26 bis rue Jeanne d'Arc 94160 Saint Mandé

01 53 66 95 28

Rédacteur en Chef : Clad Strife

Conception graphique : Weel

Illustrations : Lechatkitu

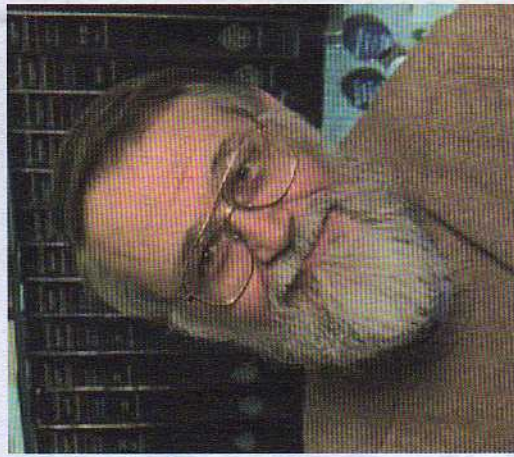
imprimé aux imprimeries de Champagne

Directeur de la Publication : Olivier Spinelli

© DMP

ENTREZ DANS LA DANSE...

Le langage C est à son origine un langage développé dans les laboratoires Bell, à partir de 1972, par un ingénieur talentueux : Dennis M. Ritchie. En 1978, les compères Ritchie et Brian W. Kernighan définissent ensemble ce que devra être le langage C. On connaît alors ce langage sous le nom de K&R C, qui est détaillé dans un livre plus que mythique : « *The C Programming Language* ». En 1983 l'institut national des standards américains (ANSI) se met à l'étude d'une standardisation de ce langage et publie, en 1988, le standard ANSI-C. Le livre de K&R est alors republié en une deuxième édition respectant ce standard.



Brian W. Kernighan travaille aujourd'hui dans les laboratoires de Bell.

Le C est un langage proche de la machine. Ce que l'on faisait jusque là en des dizaines d'instructions assembleur tient en deux lignes sous cette forme. Mais qui dit « *proche de la machine* » dit « *procédural* », soit à la fois une force et une faiblesse par rapport aux langages modernes, plus orientés objet.

Si le langage C résiste si bien au temps, c'est qu'il est très puissant, il permet de tout faire. Une fois passées les premières appréhensions sur la pseudo-complexité dudit langage, on en vient très vite à pouvoir implémenter facilement des routines de code performantes et modulaires. Et, gros avantage que vous avez à le connaître : une fois maîtrisé, il est beaucoup plus facile d'apprendre n'importe lequel... Bon courage ;-)



Dennis M. Ritchie fait également de la recherche dans les laboratoires de Bell.

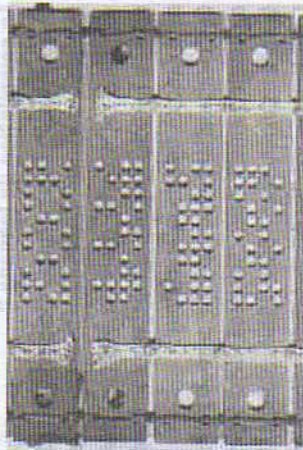
LA CHAÎNE DE COMPILE

Pour beaucoup de débutants, la compilation est un processus un peu mystique. Cet article explique en détails la chaîne de compilation des programmes C à travers l'histoire de la compilation.

Histoire de la compilation

Au début de l'histoire des ordinateurs, les opérateurs actionnaient des leviers pour coder les nombres à calculer, la compilation n'existait pas. Par la suite, le concept de programme est né : une suite d'instructions sur une carte perforée indiquant l'ordre des opérations à effectuer ; c'était donc un programme.

On procédait en écrivant un programme sur papier, on traduisait ensuite le programme en instructions sur carte perforée avec une machine à écrire spéciale. Lorsque les programmes fonctionnaient, on les archivait dans des bibliothèques pour ne pas avoir à les réécrire quand on en avait besoin - le concept de bibliothèque de programme était né.



La tâche de traduction, du programme écrit au programme sur carte perforée, était longue et sujette à erreurs. Il fallait des programmes permettant d'entrer du texte en mémoire corres-

pondant à une routine (comme ceux écrits précédemment sur papier), puis de traduire ce texte en code d'instructions.

Cette étape est aujourd'hui appelée étape d'assemblage, elle permet la traduction de code en langage machine (le code directement exécuté par le processeur).

Les langages d'assemblage sont dépendants du microprocesseur utilisé et demandent du programmeur de faire attention à beaucoup de détails. Beaucoup d'erreurs de programmation surviennent donc (des "bugs").

La première solution à ce problème fut de construire sur ce qui existait déjà, on inventa les macro-processeurs. Ces programmes permettent d'associer un nom à une routine ; dès que ce nom est trouvé dans le fichier source, il est remplacé par la routine en question.

Ce processus a évolué pour permettre de paramétrer les remplacements et de contrôler les noms auxquels les programmes ont accès.

Les méthodes de préprocesseurs sont devenues tellement évoluées que l'on avait un langage de programmation permettant d'écrire des programmes. A partir de cette observation, on inventa les langages de haut niveau comme le FORTRAN, le Pascal puis... le C.

Ces langages ont une syntaxe qui n'est pas spécifique

ressources mémoire et processeur, les modules de programmes sont distribués sous forme binaire et non sous forme de code.

Il faut donc un moyen de récupérer les morceaux de programmes contenus dans des bibliothèques binaires et les lier au programme qui l'utilise. Cette étape est réalisée grâce à un éditeur de liens.

Étant donné que beaucoup de programmes utilisent les mêmes bibliothèques, on a trouvé judicieux de faire en sorte qu'elles ne soient chargées qu'une seule fois en mémoire. Ceci permet de gagner beaucoup de place.

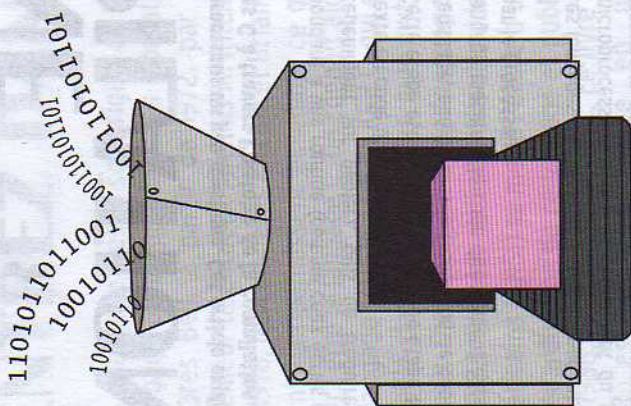
La bibliothèque est liée au programme (étape d'édition de liens) au moment du chargement du programme, de telle sorte que si elle est déjà chargée en mémoire, c'est la copie déjà chargée qui est utilisée.

Cette technique a un deuxième intérêt : les bibliothèques sont séparées des programmes, c'est-à-dire que l'on peut mettre à jour la bibliothèque sans modifier les programmes qui l'utilisent.

La compilation du langage C

La compilation du langage C comprend toutes les étapes décrites précédemment. Nous allons voir dans cette partie quels programmes sont utilisés par chacune des étapes et comment récupérer les fichiers intermédiaires.

Ces manipulations étant spécifiques aux outils de développement, nous choisissons de présenter le compilateur GCC (GNU Compiler Collection) qui est l'un des compilateurs les plus utilisés. Il est présent par défaut sur tous les UNIX, Linux, MacOSX. Il peut être installé sur Windows en téléchargeant l'environnement de développement Dev-C++ (<http://www.bloodshed.net/dev/devcpp.html>).



à un microprocesseur. On utilise ce que l'on appelle aujourd'hui un compilateur pour créer des programmes en langage d'assemblage à partir du code source dans le langage de haut niveau.

Ces langages ont en général des syntaxes plus proches des mathématiques et du langage naturel, de plus, ils laissent au compilateur le soin de faire attention aux détails de chaque microprocesseur ce qui réduit considérablement le nombre d'erreurs de programmation.

Certains langages, comme le C, ont encore un macro-processeur, on appelle ça un préprocesseur.

Lorsque les programmes augmentent en complexité, il faut pouvoir écrire des morceaux de programmes et les réutiliser lorsqu'on en a besoin.

La compilation prenant du temps, beaucoup de

GCC en réalité ne fait rien tout seul. Il appelle des programmes qui font le travail à sa place. Chaque étape de la compilation est prise en charge par un programme spécial. On peut demander à GCC de limiter son action : seulement l'un de ces programmes doit être exécuté en passant une option spéciale.

La première étape est de passer le préprocesseur sur les fichiers sources, de transformer les .c (fichiers source en langage C) en .i (fichiers source en langage C qui n'ont pas besoin d'être passés au préprocesseur). Ceci se fait avec la commande :

```
$ gcc -E fichier.c -o fichier.i
```

Le programme appelé par derrière est "cpp", le préprocesseur.

Après cette étape, chaque fichier .c contient tout le contenu des fichiers d'en-tête (include files), toutes les macros sont étendues, le code n'est plus que du C pur, ce qui permet au compilateur de le compiler.

L'étape suivante est la génération du code assembleur à partir du source C. Pour faire ceci, entrez la commande :

```
$ gcc -S fichier.c
```

Le programme appelé par derrière est "cc1", le compilateur.

Ceci produit le fichier "fichier.s" contenant le code assembleur pour votre micro-processeur.

La prochaine étape est un peu complexe. Il faut générer un fichier objet à partir du fichier assembleur. Comme expliqué dans la première partie, on veut pouvoir créer des bibliothèques

avec des morceaux de programmes.

Les briques de base de ces bibliothèques sont les fichiers objets. Un fichier objet est la version compilée des fonctions qui se trouvaient dans le fichier source, c'est à dire une sorte de mini-bibliothèque.

```
$ gcc -c fichier.s
```

il est possible de passer directement le fichier C

```
$ gcc -c fichier.c
```

Le programme appelé par derrière est "as", l'assembleur.

Ceci produit le fichier "fichier.o" contenant le code machine du programme compilé. A partir du fichier objet, on peut créer une bibliothèque en lançant la commande :

```
$ ar cr libfichiers.a fichier1.o \
    fichier2.o fichier3.o ....
$ ranlib libfichier.a
```

Ceci crée le fichier `libfichier.a` qui est un biblio-

A partir du fichier objet, on peut aussi créer un exécutable, via la commande :

```
$ gcc fichier1.o fichier2.o \
-o mvexec
```

ci, on crée le fichier "myexec" qui est exécutable. Si l'un des fichiers objet utilise une fonction d'une bibliothèque, il faut l'indiquer sur la ligne de commande avec l'option "-l<nom-lib>" et "-l<répertoire-lib>", exemple :

```
$ gcc fichier1.o fichier2.o
-lfichier -L. -o myexec
```

RÈGLES DE PROGRAMMATION EN C

Programmer en C, c'est bien, programmer proprement en C, c'est mieux. Les règles suivantes vous aideront à créer un code lisible et compréhensible par tous, et à éviter quelques erreurs bêtes.

Instructions et commentaires

Les instructions composent l'essentiel du programme. Il est donc important qu'elles soient lisibles. Aussi, la règle est : une seule instruction par ligne (si possible, suivi d'un commentaire explicite sur son rôle).

Les commentaires sont très importants : ils permettent aux programmeurs (auteurs du code ou non) de ne pas avoir à se casser la tête pour savoir ce qu'un bout de code fait. Que vous travailliez en équipe ou non, commentez toujours votre code, aussi court soit-il.

Il existe deux sortes de commentaire en C :

- les commentaires multi-lignes, qui sont placés entre /* et */
- les commentaires sur une ligne, qui sont placés entre // et la fin de la ligne.

Le mieux consiste à placer les commentaires généraux portant sur un ensemble d'instructions sur des lignes isolées, avant l'instruction concernée, tandis que les commentaires précisant un détail seront placés à droite de l'instruction en question.

Exemple

```
/* fonction isZero :
 * retourne 1 si l'entier passe en
 * argument est nul, 0 sinon
 */
int isZero(int n)
{
    if (n==0) //si l'entier est nul
        return 1;
```

La compilation de programmes C est donc un processus long et complexe, même si ça n'en a pas l'air lorsqu'on ne se préoccupe pas des étapes et fichiers intermédiaires.

Certaines étapes intermédiaires sont cependant importantes à retenir, en particulier lorsque l'on doit gérer les fichiers source de plus gros projets.

Pour ceux qui ne lancent pas leurs commandes de compilation dans un shell (je pense aux utilisateurs de Windows ou autre environnement de développement graphique), toutes les options présentées restent disponibles à travers les menus de configuration.

Philippe Amarengo

Ceci assemble le fichier exécutable "myexec" à partir des fichiers objets "fichier1.o" et "fichier2.o", dont l'un utilise la bibliothèque libfichier.a contenue dans le répertoire "." (répertoire courant).

Notez que si le fichier contenant la bibliothèque s'appelle libfichier.a, il faut passer -lfichier au compilateur et non pas -libfichier.a.

Lorsque les étapes intermédiaires ne vous intéressent pas, vous pouvez très bien passer directement les fichiers C en argument à gcc :

```
$ gcc fichier1.c fichier2.c
-lfichier -L. -o myexec
```

Un petit résumé

Le langage C est un langage compilé. Le fichier source ne peut être directement exécuté par le microprocesseur. Il doit subir des modifications d'un programme annexe (le compilateur) afin d'être traduit dans un langage de plus bas niveau directement compréhensible par le microprocesseur, ce qui permettra à notre fichier final de s'exécuter sur l'ordinateur de manière autonome.

Notre compilateur va donc devoir lire puis transformer notre fichier source. La compilation d'un programme en C se décompose en quatre parties.

La première étape est la lecture du fichier source par le pré processeur. Celui-ci va donc se charger, après lecture du code, d'inclure les headers (#include) dans le fichier source. La seconde étape est la compilation. Le fichier source précédent est transformé en assembleur, une suite d'instructions associées à des fonctions du processeur. Puis se produit l'assemblage, notre fichier subit encore une modification. Le code assembleur est alors converti en langage machine dans un fichier binaire. Ce fichier est aussi appelé fichier objet. Vous pensiez que c'était terminé et pourtant non, le compilateur appelle alors le linker qui va se charger d'intégrer tous les éléments extérieurs comme les librairies et les fonctions pour créer notre dernier fichier exécutable, et ce de façon autonome.

```
else //sinon, il n'est pas nul ;)
    return 0;
}
```

Autre chose, évitez les goto. Ils rendent en effet plus difficile la compréhension du code par d'autres personnes, qui doivent à chaque fois retrouver l'étiquette correspondante et la logique sous-jacente. Remplacez-les par des boucles, plus claires et plus faciles à lire.

Enfin, évitez également le « C condensé » : remplacez cinq lignes par une ne sert à rien d'autre qu'à compliquer la relecture, y compris la vôtre. Les deux codes suivants sont équivalents, mais le second est nettement plus compréhensible :

```
x=((a==4)?++i:tab[j++])*c--;
//en clair :
if (a==4)
{
    i++;
    x = i;
} else {
    x = tab[j];
    j++;
}
x = x * c;
C--;
```

Bien sûr, ce sont ici deux exemples extrêmes, et vous n'êtes pas obligés de simplifier au maximum. Ainsi, ce qui suit est assez lisible :

```
if (a==4)
    x = ++i;
```

```
else
```

```
  x = tab[j++];
  x = x * C--;
```

Indentation

L'indentation sert à rendre le code plus lisible par l'insertion d'espaces avant les instructions. Chaque instruction d'un bloc d'instructions (boucle, fonction, définition) doit être précédée par le même nombre d'espaces. Lorsqu'on a des blocs imbriqués (les uns dans les autres), on augmente de manière régulière le nombre d'espaces (par exemple deux espaces à chaque fois).

Exemple

```
//definition
struct maStructure
{
    int var1;
    char var3[10];
}

//fonction
int test_fonction (void)
{
    instruction1;
    instruction2;
    instruction3;
    //bloc if dans la fonction
    if (test)
    {
        instruction4;
        instruction5;
    }
}
```

Lorsqu'un bloc d'instructions n'en contient qu'une seule, elle est quand même indentée :

```
if(a==4)
    printf("a = 4 !");
```

Si votre code est correctement indenté, il sera beaucoup plus facile à relire, pour vous comme pour les autres.

Remarque : il est bien question d'espaces, et non de tabulations. En effet, tous les éditeurs n'affichant pas les tabulations de la même façon (plus ou moins grandes), il est déconseillé de les utiliser. La plupart des éditeurs proposent d'insérer un nombre défini d'espaces à la place des tabulations. De même, utilisez toujours une police à pas fixe (tous les caractères ont la même taille), par exemple Courier New.

Accolades

Les accolades marquent le début et la fin d'un bloc d'instructions. Il est conseillé de les placer seules sur une ligne, et de les indenter comme ce qui les précède. Ainsi, vous risquez moins de vous tromper dans le nombre d'accolades et retrouverez rapidement une accolade manquante.

Exemple

```
if (test)
{
    instruction1;
    instruction2;
}
```

Certaines personnes préfèrent mettre l'accolade ouvrante sur la même ligne que ce qui la précède :

```
if (test) {
    instruction1;
    instruction2;
}
```

Il vaut mieux n'utiliser cette forme que dans les courts exemples où aux endroits où la place est

limitée, dans un article par exemple .p (Les conseils n'engagent que l'auteur ;-) (NDLR))

Pointeurs

Un truc tout bête, mais qui peut parfois poser un problème : la position de l'étoile dans la déclaration et l'utilisation de pointeurs. En effet, si je déclare :

```
int *a, b;
int* c, d;
```

de quels types sont b et d ? Entier, oui, mais ce n'est pas flagrant. Pour éviter toute confusion, beaucoup de développeurs mettent la ou les étoiles entre espaces :

```
int * a, * b; //a et b sont des pointeurs sur entiers
```

De même, certains préfèrent mettre un seul pointeur par ligne, et font ainsi grâce au début tant de l'erreur classique « l'étoile se propage-t-elle ? » :

```
int * a;
int * b; //rien de plus clair, non ?
```

Noms de variables

Peut-être certain d'entre vous ont-ils entendu parler de la notation hongroise. Proposée par Charles Simonyi, elle consiste à commencer le nom des variables par une ou deux lettres correspondant à son type.

Exemple

```
int iMonEntier1; // i pour int
// sz pour "String Zero" (chaîne
// finissant par le caractère nul)
char szMaChaine1;
float fMonFlottant; //etc
```

Cela permet entre autres d'éviter les confusions de type. Personnellement, je n'utilise pas toujours cette notation, car elle est parfois un peu lourde. En revanche, il est impératif que le nom d'une variable (ou d'une fonction) reflète son rôle.

Exemple

```
int nbTours; // nombre de tours
float prixObjet; // prix d'un objet
// dessine la map dans la bitmap
// passe en argument
void drawMap(BITMAP * map);
```

Un exception est faite pour certains noms couramment utilisés :

i, j pour des compteurs (notamment dans des boucles for),
fp, fd, fdesc pour un descripteur de fichier,
x, y, z pour des inconnues mathématiques ou des coordonnées,
n pour un entier quelconque.

Évitez les mots entièrement en majuscules et les underscores, qui sont, de préférence, utilisés pour les constantes et les macros. (NDLR : de façon générale, il vaut mieux ne pas mettre de majuscules dans les noms de variables et de fonctions en C. Ce genre de pratique est plutôt réservé à des langages de haut niveau tels le C++, le Java... Mais ce ne sont que des conventions.)

Constantes et macros

Une convention astucieuse consiste à écrire les noms de constantes et de macros à l'aide de majuscules et de blancs soulignés (underscore : '_'), et à n'utiliser ces caractères dans aucun autre cas.

Exemple

```
#define _NB_MUNITION_MAX_ 1000
#define _ERREUR_(msg){\
    printf("Erreur : %s", msg);\
    return 1;}
```

Plus le nom est explicite et spécifique au rôle de la constante, moins il y a de chance pour qu'il soit réutilisé (ou déjà utilisé) pour autre chose. Autre possibilité : faire précéder tous les noms de constantes par le nom du projet ou de la section.

Exemple

```
#define _PROG1_FILE_MAX_SIZE_
65535
```

Fichier .c et .h

Excepté pour un programme très court, il est déconseillé de tout mettre dans un seul fichier, car il devient difficile de se repérer parmi plusieurs fonctions.

De plus, si vous comptez réutiliser une partie de vos fonctions (par exemple celles concernant le réseau), il est plus pratique de copier un fichier que de copier-coller une dizaine de fonctions éparpillées dans un fichier de quelques centaines de lignes.

On regroupe donc les fonctions par domaine dans des fichiers .c que l'on pourra réutiliser par la suite dans d'autres projets. On obtiendra donc, par exemple, `reseau.c`, `graphiques.c`, `divers.c` et ainsi de suite. Ces fichiers ne sont pas à inclure par un `#include`, mais à compiler séparément et à lier dans votre exécutable final. La plupart des environnements de développement le font automatiquement, il suffit d'ajouter les fichiers .c à votre projet. Pour les autres, je vous renvoie donc à la documentation de votre environnement et/ou de votre compilateur.

Les fichiers d'en-tête (.h), quant à eux, contiennent dans l'ordre :

- un commentaire signalant le nom et l'adresse mail de l'auteur, la date de dernière modification (et éventuellement un numéro de version) ainsi qu'un bref résumé de l'utilité du

fichier,

- les includes,
- les constantes et les macros (`#define`),
- les définitions de type (`typedef`),
- les définitions de variables (structures, déclarations de variables globales externes),
- les prototypes de fonctions,
- des commentaires décrivant le tout en détail.

Pour plus de souplesse, il est également conseillé de faire un .h par .c, contenant les prototypes des fonctions et les autres définitions essentielles au bon fonctionnement de celles-ci. Cela permet de copier juste un .c et un .h si on veut réutiliser tout un ensemble de fonctions.

Si certaines définition sont requises par plusieurs .c, les rassembler dans un .h commun avec un nom explicite, du genre « *commun.h* » (attention « *common.h* » est souvent déjà utilisé).

Chaque .c appelle tous les .h nécessaires avec `#include`, sans se soucier des inclusions multiples car celles-ci seront évitées par l'astuce expliquée plus loin.

Le main se trouve dans le fichier principal du programme, que je vous conseille d'appeler `main.c` pour le reconnaître facilement. Dans ce fichier, on inclut donc un `main.h`, qui se charge d'inclure tous les autres .h afin que le main puisse appeler les autres fonctions (grâce à leurs prototypes).

VARIABLES GLOBALES

Dans la mesure du possible, il faut éviter d'utiliser des variables globales, car elles peuvent entrer en conflits avec les variables locales de chaque fonction. Cependant, si vous souhaitez vraiment les utiliser, le mieux est de les déclarer dans le `main.c`, et faire une déclaration externe dans le `main.h`, et évidemment de leur donner un nom qui se retrouve difficilement dans d'autres fonctions (une variable globale peut commencer par « *gl_* »).

Exemple

```

dans main.h :
extern int global_monEntier;

dans main.c :
int global_monEntier;
```

Remarque : si une variable globale n'est pas utilisée par le main, il est courant qu'elle soit déclarée dans un des .c qui contiennent les autres fonctions utilisant cette variable.

Protection des fichiers d'en-tête

Les fichiers d'en-tête (.h) sont souvent inclus plusieurs fois. Aussi, pour éviter les erreurs de type « *déclaration multiple* », il faut les protéger. La manière la plus simple consiste à placer le contenu de votre fichier entre `#ifndef` et `#endif`. Cela permet au compilateur de n'utiliser ce contenu que si une certaine constante n'est pas (déjà) définie. Dans ce contenu, on ajoute la définition de cette constante avec `#define`. Ainsi, une fois que le fichier a été inclus une fois, la constante est définie, et si le fichier est inclus plusieurs fois, le compilateur ne prendra plus en compte le contenu.

Exemple :

```

entete.h
#ifndef _ENTETE_H_INC_
#define _ENTETE_H_INC_

// contenu du fichier entete...

#endif
```

A vous de jouer

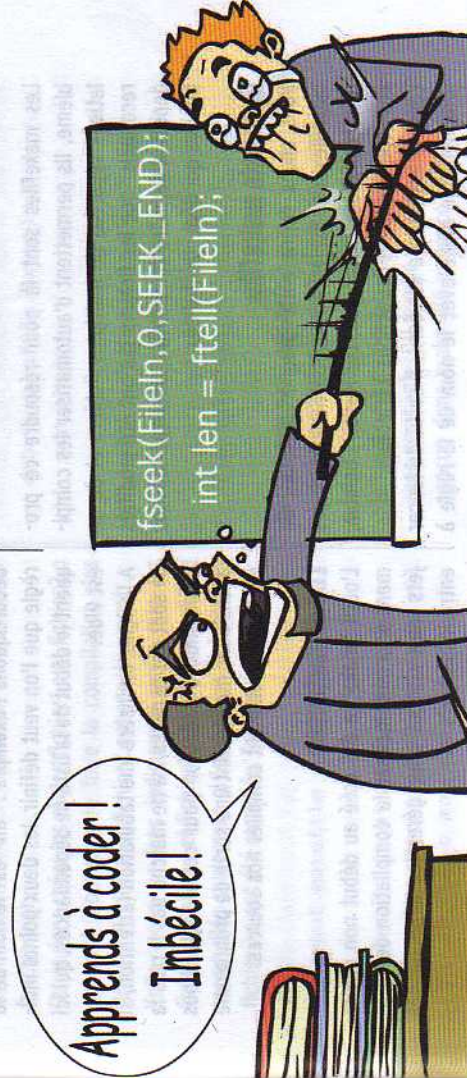
Bien entendu, les règles énoncées ci-dessus ne sont pas absolues. Ce sont celles qui me conviennent et que j'utilise, dont certaines sont communes à pas mal de programmeurs. C'est à vous seul de décider lesquelles vous voulez suivre, et d'en inventer d'autres que vous jugez utiles. N'hésitez pas à en discuter sur les forums consacrés au C ou au développement en général, afin de les confronter à celles d'autres codeurs et d'en apprendre davantage. Cependant, quelles que soient les règles que vous vous fixez, le principal est de vous y tenir, et de les utiliser pour tous vos codes.

« C'est la régularité qui donne la lisibilité. » (FAQ-FCCLC)

Merci à la FAQ-FCCLC :

<http://www.istty.info.uvsc.fr/~rumeau/fc/c/> et à Olivier L.

ek



APPRENEZ À ÉCRIRE UN MAKEFILE

On a beau avoir tapé "make && make install" des centaines de fois, on ne sait peut-être toujours pas ce qui se passe sous le voile des lignes qui défilent...

Cet article présente une manière d'automatiser la compilation. Si vous n'avez pas lu l'article sur la chaîne de compilation, c'est le moment de vous y mettre car la connaissance des procédures de compilation est pré-requis pour aborder cet article.

Concepts de makefile

Lorsqu'un projet commence à grossir à plusieurs fichiers, il devient fastidieux de taper la commande de compilation pour chacun d'eux. De même, il est inutile de recompilier chaque fichier alors que seuls les fichiers modifiés ont besoin de l'être. Car autant la compilation d'un fichier est rapide, autant la compilation de plusieurs fichiers rend la consommation de ressources non négligeable.

Lorsque le projet atteint une dizaine de fichiers de bonne taille, il devient indispensable de se doter d'un système de compilation automatique.

Les makefiles sont là pour résoudre ce problème. Ils permettent d'automatiser les compilations séparées de multiples fichiers et de ne recompiler que les fichiers qui ont été modifiés depuis la dernière compilation ainsi que tous les fichiers qui dépendent d'eux.

Un simple makefile

Nous allons écrire notre premier makefile. Un makefile est un fichier texte qui contient des règles de compilation. Chacune des règles a un nom, des dépendances et des actions à effectuer.

Lorsque le makefile est écrit, il suffit de lancer la commande make avec le nom de la règle à

effectuer pour que make l'exécute. Si aucune règle n'est donnée sur la ligne de commande, la règle "all" est exécutée.

```
— debut makefile —
all:
    gcc source.c -o myexec
— fin makefile —

$ make
ou
$ make all
```

Décrivons l'exemple : "all" est le nom de la règle que l'on veut définir, les deux points indiquent le début de la liste de dépendances, qui ici est vide.

A la ligne, on insère une tabulation (attention, si ce sont des espaces, ça ne marche pas !) puis la commande à exécuter pour cette règle. Nous avons donc maintenant une sorte de programme expliquant comment compiler nos sources.

Les dépendances

L'outil "make" a été créé au début non seulement pour automatiser la compilation des projets mais surtout pour gérer les dépendances entre les fichiers.

En effet, lorsque vos projets commencent à devenir un peu gros, c'est à dire trois ou quatre fichiers, il devient long de tout recompiler à chaque fois que l'on veut tester que notre code compilé fonctionne. Il nous faut donc une manière de ne recompiler que les fichiers qui ont été modifiés.

Make permet de le faire. Il faut utiliser le fait que la compilation se fait en plusieurs étapes et que les étapes les plus longues peuvent se faire séparément pour chaque fichier. Pour chaque fichier C, on va d'abord créer un fichier objet, puis quand on les aura tous, on créera l'exécutable à partir de ça. Si l'on écrivait à la main les lignes dans un terminal, ça serait :

```
$ gcc -c fichier1.c -o fichier1.o
$ gcc -c fichier2.c -o fichier2.o
$ gcc fichier1.o fichier2.o -o mon_appli
```

On va créer la règle mon_appli qui va s'occuper de faire l'étape d'édition de liens pour nous. On va dire à make que mon_appli a besoin des fichiers fichier1.o et fichier2.o pour pouvoir fonctionner.

On va donc créer les règles fichier1.o et fichier2.o qui vont s'occuper de faire la compilation des fichiers fichier1.c et fichier2.c respectivement. Enfin, nous voulons indiquer à make que les fichiers fichier1.o et fichier2.o n'ont besoin d'être recompilés seulement si les fichiers fichier1.c et fichier2.c ont été modifiés. Tout ceci s'écrit :

```
— debut makefile —
mon_appli: fichier1.o fichier2.o
    gcc fichier1.o fichier2.o -o mon_appli
```

```
fichier1.o: fichier1.c
    gcc -c fichier1.c
fichier2.o: fichier2.c
    gcc -c fichier2.c
— fin makefile —
```

Vous pouvez ensuite taper dans un terminal (ou cliquer sur "compiler" dans un éditeur graphique) :

```
$ make mon_appli
```

Les variables et règles génériques

Vous savez maintenant faire des makefiles pour vos projets. Vous avez remarqué cependant que, lorsque l'on doit gérer un grand nombre de fichiers, il devient fastidieux d'écrire toutes les règles pour chaque fichier.

De plus, on remarque que la règle pour passer d'un fichier .c à un fichier .o est toujours la même au nom du fichier près. On aimerait donc avoir un moyen de spécifier des règles indépendantes du nom du fichier.

Ceci est possible grâce à une directive et quelques variables spéciales définies par make. Ça s'écrit comme ceci (exemple 1) :

```
— debut makefile exemple 1 —
all: mon_appli
    mon_appli: fichier1.o fichier2.o
    gcc fichier1.o fichier2.o -o mon_appli
— fin makefile —
```

Décrivons cet exemple :

- La règle all dépend de la règle mon_appli.
- La règle mon_appli dépend des règles fichier1.o et fichier2.o, make va donc chercher ces règles. Il tombe sur une règle pré-conçue dans make qui annonce être capable

de convertir les .c en .o et voit donc l'opportunité de récupérer des .o.

- Il demande à cette règle de lui donner les .o correspondants aux fichier1.o et fichier2.o. Cette règle dit qu'elle ne génère des .o qu'à partir des .c correspondants et en plus seulement si le fichier .c a été modifié depuis la dernière fois que le .o a été généré.
- Les fichiers .o sont générés, la règle mon_appli peut être accomplie et enfin la règle all.

Le code du Makefile est beaucoup plus court, beaucoup moins redondant, et plus facile à lire.

On remarque que ceci ne répond pas entièrement à notre problème. En effet, il faut écrire deux fois la liste des fichiers .o ce qui est redondant et source d'erreur.

De plus, on aimerait avoir des règles bien séparées des données de telle sorte que l'on ne change que la liste des fichiers à compiler. Pour ceci, on utilise des variables.

En langage Makefile, on assigne une valeur à une variable avec l'opérateur "=". On récupère le contenu d'une variable avec \$(nom_de_la_variable). Le but était de ne pas réécrire deux fois la liste des fichiers .o et de n'avoir qu'à ajouter un fichier .o dans la liste lorsque l'on ajoute un fichier au projet. Ajoutez / modifiez ces règles dans le makefile pour inclure ces fonctionnalités (exemple 2).

```

— debut makefile exemple 2 —
OBJECTS = fichier1.o fichier2.o

mon_appli: $(OBJECTS)
    gcc $(OBJECTS) -o mon_appli

— fin makefile —

```

Règles de dépendance avancées

Cette partie s'attache à montrer comment arriver à un système de dépendance entre les fichiers quasi parfait. Vous avez remarqué que les règles pour les fichiers .o avaient en dépendance le fichier .c associé.

Il n'est pas obligatoire de se limiter à ce fichier, on peut en mettre autant que l'on veut. En particulier, on peut mettre les fichiers .h en dépendance de telle sorte que si le .h est modifié alors le .c équivalent sera recompilé pour prendre en compte les modifications.

Pour faire ça, il suffit d'écrire une règle par fichier et de mettre les bons fichiers en dépendance. La règle générique est utilisée pour la compilation et les règles spécifiques sont utilisées pour calculer les dépendances. Cette technique fonctionne mais elle nous ramène à la technique que l'on utilisait au début qui nous obligeait à faire une règle par fichier.

Pour parer à ce problème, nous allons utiliser l'utilitaire "makedepend". C'est un programme qui analyse des fichiers source en C et qui génère une liste de règles de dépendance. Nous n'avons donc pas à le faire à la main. Ce programme prend en argument la liste des fichiers .c. Il faut donc une variable du makefile qui contienne la liste des fichiers source.c.

Vu que l'on a déjà la liste des fichiers .o, on se dit qu'à une extension près, on n'a pas envie de réécrire toute la liste des fichiers et on aura raison. Make, t'es là pour ça ! Nous allons utiliser une syntaxe de make qui nous permette de changer les extensions des fichiers, tout simplement.

Voici le makefile contenant toutes les fonctionnalités décrites jusque là (exemple 3) :

```

— debut makefile exemple 3 —
# makefile d'exemple
# utilisation:
# "make" compile tout le projet
# "make depend" ajoute la liste des
# dépendances au makefile

APPLI = mon_appli
SOURCES = fichier1.c \
          fichier2.c
OBJECTS = $(SOURCES:.c=.o)

all: $(APPLI)

```

```

$(APPLI): $(OBJECTS)
    gcc $(OBJECTS) -o $(APPLI)

depend:
    makedepend -f$(SOURCES) < /dev/null
— fin makefile —

```

La règle "depend" crée la liste des dépendances et l'ajoute à la fin de votre makefile. Attention à ne pas toucher à ce que makedepend a ajouté sinon il ne sera peut-être plus capable de mettre à jour cette liste.

Passage d'options de compilation

Une bonne façon d'organiser ses fichiers consiste à placer le répertoire contenant les en-têtes de vos fichiers .c (les fichiers .h) dans un répertoire "include". On peut ainsi avoir l'arborescence :

```

./proj/src/ contient les .c et le Makefile
./../proj/src/include contient les .h

```

A ce moment, make ne sait pas nativement où aller chercher les fichiers. En réalité, c'est GCC qui ne sait pas où aller chercher les fichiers .h.

Il faut préciser dans votre Makefile l'argument -I/include pour que GCC aille chercher les fichiers dans le répertoire mentionné. Les fichiers d'en-tête n'étant utilisés que lors de la compilation des fichiers sources .c en .o, une règle CFLAGS que make comprend peut nous aider (exemple 4).

```

— debut makefile exemple 4 —
# makefile d'exemple
# utilisation:
# "make compile" tout le projet
# "make depend" ajoute la liste
# des dépendances au makefile

SOURCES = fichier1.c \
          fichier2.c
OBJECTS = $(SOURCES:.c=.o)
CFLAGS = -I../include
APPLI = mon_appli

all: $(APPLI)
$(APPLI): $(OBJECTS)
    gcc $(OBJECTS) -o $(APPLI)

— fin makefile —

```

Dans cet exemple, nous avons rajouté quelques options qui seront passées à GCC lors de l'exécution du jeu de dépendances "\$(APPLI): \$(OBJECTS)".

Sachez que tous les projets utilisent make dans le monde UNIX et beaucoup sur Windows. Il faut savoir aussi que si vous utilisez un éditeur graphique, vous n'aurez en général pas besoin d'écrire vos makefile car votre éditeur le fait pour vous.

Philippe Amarengo

LES POINTEURS

Prenez le temps de faire une pause, et de lire ce qui suit à tête reposée. Parce que la notion que nous allons aborder maintenant n'est pas facile, mais essentielle à la programmation en langage C (et dans quelques autres langages aussi).

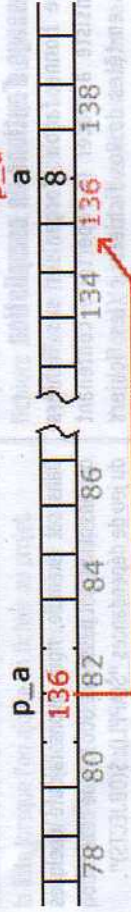
Adresse d'une variable

Les variables du programme sont stockées dans la mémoire vive durant l'exécution de celui-ci. La valeur d'une variable est donc située à un endroit dans la mémoire : cet endroit s'appelle adresse (ou adresse mémoire) de la variable.

Un pointeur est une variable qui contient une adresse mémoire, c'est-à-dire une adresse où se trouve, par exemple, le contenu d'un entier. On dit qu'un pointeur pointe sur une variable s'il contient l'adresse mémoire à laquelle est stocké le contenu de la variable. Les pointeurs permettent donc de manipuler des données en utilisant non pas leur nom, mais leur adresse.

L'utilisation du nom de la variable pour accéder à son contenu s'appelle l'adressage direct, l'utilisation d'une adresse mémoire, l'adressage indirect.

Le pointeur `p_a` contient l'adresse de `a`.



L'opérateur &

L'opérateur « & » permet d'obtenir l'adresse de la variable devant laquelle il est placé. &a est l'adresse de `a`. Cette adresse peut être stockée dans un pointeur, ou passée en paramètre à une fonction (qui attend un pointeur en argument, voir plus loin).

Exemple

```
int a;  
int * p_a;  
p_a = &a; // p_a contient maintenant l'adresse de a
```

ADRESSAGE INDIRECT

En précédant le nom du pointeur par l'opérateur étoile, on peut manipuler non pas le contenu du pointeur (l'adresse de la variable), mais directement le contenu pointé, c'est-à-dire le contenu de la variable pointée.

Exemple

```
*p_a = 8; // qui est de l'adressage indirect  
// qui est équivalent à : a = 8; qui est lui-même de l'adressage direct
```

Manipuler `*p_a` revient donc exactement au même que manipuler `a`, puisqu'on manipule le contenu situé à l'adresse `p_a`, qui est l'adresse de `a` (voir schéma précédent). L'utilité d'une telle manœuvre va vous apparaître par la suite.

Avec des fonctions

Lorsqu'on passe des variables en arguments à une fonction, celle-ci reçoit une copie de ces variables, donc ne peut modifier l'original. Pour modifier malgré tout une variable à travers une fonction, il suffit de passer son adresse (donc un pointeur), et de modifier le contenu situé à cette adresse (pointé par le pointeur). La fonction reçoit donc un pointeur sur la variable.

Exemple : modification de la valeur d'un entier

```
void ajoute_2(int * p_a) {  
    *p_a += 2;  
}  
// pour ajouter 2 à l'entier a :  
ajoute_2(&a);
```

On passe donc l'adresse de `a` (&a) à la fonction, qui ajoute 2 au contenu situé à cette adresse, donc on ajoute bien 2 à l'entier `a`.

C'est exactement ce que vous faites lorsque vous utilisez la fonction `scanf`, qui prend comme argument l'adresse des variables à modifier :

```
scanf("%d", &a); // affecte a la valeur entrée par l'utilisateur
```

Exercice : faire une procédure qui permet de modifier un entier, en lui ajoutant une valeur variable (passée en 2^e argument).

Allocation dynamique de mémoire

Une autre utilisation courante des pointeurs est l'allocation dynamique de mémoire, qui permet de créer des variables au milieu d'un programme, sans les avoir déclarées explicitement au préalable. Le principe est le suivant : au début du programme (ou de la fonction), on déclare juste un pointeur vers un type de variable que l'on souhaite créer (par exemple `int`). On demande un espace de la mémoire suffisant pour ce type puis on fait prendre au pointeur

une nouvelle valeur : l'adresse de l'espace mémoire ainsi réservé.

Ces deux dernières étapes se font à l'aide de la fonction `malloc`. Celle-ci prend en argument la taille que l'on souhaite obtenir dans la mémoire, et retourne un pointeur vers l'espace ainsi créé, ou `NULL` en cas d'échec.

La taille d'un type de variable s'obtient à l'aide de la fonction `sizeof` : `sizeof(int)` renvoie la taille d'un entier, `sizeof(char)` la taille d'un caractère, etc.

Exemple :

```
int main(void) {  
    // déclaration d'un pointeur sur  
    // entier  
    int * p_entier;  
    // diverses instructions...  
    // réservation d'un espace mémoire  
    // et attribution de son adresse  
    // au pointeur  
    p_entier = (int *) malloc(sizeof(int));  
    // attribution de la valeur 2 à  
    // l'entier  
    *p_entier = 2; // puis on lui donne une  
    // libération de la mémoire allouée  
    free(p_entier);  
    return 0;  
}
```

Il ne nous reste qu'à faire précéder le nom du pointeur par une étoile.

Ne pas oublier de libérer la mémoire allouée dynamiquement avec la fonction `free`, même juste avant la fin du main.

ek

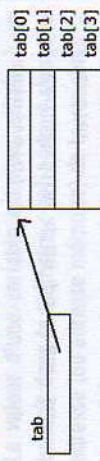
Greetz : Olivier L.

TABLEAUX DE POINTEURS

Les tableaux en C sont souvent une source de confusion, même chez des programmeurs initiés, qui pourtant en usent et en abusent.

Tableaux et pointeurs

Dans beaucoup (trop) de cours et faq sur le C, on peut lire qu'un tableau est un pointeur qui pointe sur un espace mémoire contenant les éléments les uns à la suite des autres. Ce qui donnerait le schéma suivant :



Où `tab` pointerait sur le premier élément du tableau, c'est-à-dire que `tab = &tab[0]` ; En réalité, un tableau n'est pas un pointeur : un tableau est une zone mémoire susceptible de contenir plusieurs éléments consécutifs de même type, alors qu'un pointeur est une variable contenant l'adresse d'une autre variable. Cela dit, en pratique, ils se comportent (et s'utilisent) presque tout le temps de la même manière. Vous pouvez donc retenir le schéma précédent, à condition de ne pas oublier que ce n'est pas exactement un tableau, mais son quasi-équivalent.

Exemple

Comme vous le savez peut-être, une chaîne de caractères est un tableau (ou un pointeur sur une suite) de caractères, dont le dernier est le « caractère nul », noté `\0`.

On peut donc allouer dynamiquement une chaîne de caractères ainsi :

```
//déclaration d'un pointeur sur
caractere char * chaîne;

chaîne = (char *) malloc(
    51 * sizeof(char));
```

de caractères avant le `\0`, donc le nombre d'octets moins 1 :

```
printf("%d %d\n", strlen(tab)+1,
    strlen(ptr)+1);
```

affichera : 8 6

Cependant, pour les tableaux dynamiques contenant autre chose que des caractères, il faudra une variable supplémentaire pour noter le nombre d'éléments.

Remarque : Certains compilateurs refusent de compiler un programme où l'on déclare un tableau sans dimension :

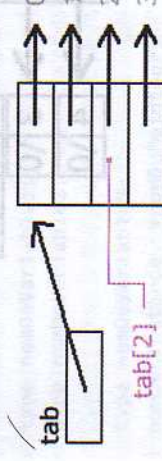
```
int a[]; // provoque l'erreur :
```

```
array size missing in 'a'
```

Car a n'étant pas un pointeur mais un tableau sans dimension, il ne peut servir à rien et ne peut même pas exister. On utilise donc un pointeur `int * a`. D'autres compilateurs, plus laxistes, laisseront passer cette déclaration en considérant a comme un pointeur. En revanche, il est tout à fait possible d'utiliser les crochets vides dans une définition ou une déclaration de fonction, comme nous allons le voir.

Tableaux de pointeurs

Puisqu'on peut déclarer dynamiquement un équivalent à un tableau, on va faire de même avec les tableaux à plusieurs dimensions. Il nous faut donc plusieurs pointeurs, qui pointent chacun vers un tableau dynamique, et que nous regroupons dans un tableau de pointeurs, selon le schéma suivant :



Remarque : ceci n'est pas un « tableau multidimensionnel », car ce nom désigne un tableau continu en mémoire (or ici, ce n'est pas nécessairement le cas). Si on voulait un tableau multidimensionnel, il faudrait utiliser :

```
int tbl[nb_lignes * nb_colonnes];
```

et accéder au tableau, non pas par `tbl[i][j]` mais par :

```
tbl [ i * nb_lignes + j ]
```

ou `i` est le numéro de la ligne et `j` celui de la colonne (les numéros commençant toujours à 0).

Exemple : création d'un tableau de pointeurs (sur entiers) de `nb_lignes` lignes et `nb_colonnes` colonnes, équivalent à la déclaration statique :

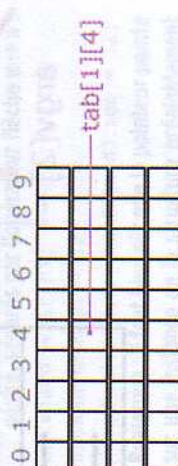
```
int tab[nb_lignes][nb_colonnes];
```

```
#include <stdlib.h>
int main(void) {
    int ** tab; //tab est un pointeur sur pointeur sur entier
    int i;

    //creation du tableau de pointeurs
    tab = (int **) malloc(nb_lignes
        * sizeof(int *));

    //affectation de chacun des
    //pointeurs a un tableau
    //dynamique (cree en meme temps)
    for(i = 0; i < nb_lignes; i++)
        tab[i] = malloc(nb_colonnes *
            sizeof(int));

    return 0;
}
```



LES STRUCTURES

Il arrive souvent que nous ayons besoin de rassembler plusieurs informations dans un même tout, afin d'éviter une profusion de variables isolées. Les tableaux, notamment, nous permettent de manipuler plusieurs données de même type. Les structures permettent de manipuler plusieurs données de types différents.

Définition

La définition d'une structure se fait à l'aide du mot-clé `struct`, de la manière suivante :

```
struct nomDeLaStructure
```

```
{
    type nomDeVariable1;
    type nomDeVariable2;
    ...
    type nomDeVariableN;
};
```

Attention à ne pas oublier le point-virgule en fin de définition.

Déclaration

Une fois définie, on peut créer une instance de la structure (c'est à dire déclarer une variable dont le type est cette structure) ainsi :

```
struct nomDeLaStructure nomDeL'instance;
```

Nous pouvons également créer un pointeur sur cette structure :

```
struct nomDeLaStructure *nomDuPointeur;
```

Comme pour une déclaration de variable, la déclaration peut être multiple :

```
struct nomDeLaStructure nomDeL'instance
*nomDuPointeur;
```

Rien ne vous empêche de faire la définition et la déclaration ensemble :

```
struct nomDeLaStructure
```

```
{
    type nomDeVariable1;
    type nomDeVariable2;
    ...
    type nomDeVariableN;
} nomDeL'instance, *nomDuPointeur;
```

On obtient un tableau de pointeur équivalent à un tableau à deux dimensions : l'accès aux éléments se fait donc via `tab[i][j]` où `i` est le numéro de ligne, et `j` le numéro de colonnes (les numéros commencent toujours à 0).

Remarque : Autre différence avec les tableaux réels, on peut ici faire des lignes de tailles différentes, il faut alors les faire une à une :

```
tab[0] = malloc(4 * sizeof(int));
tab[1] = malloc(122 * sizeof(int));
...
```

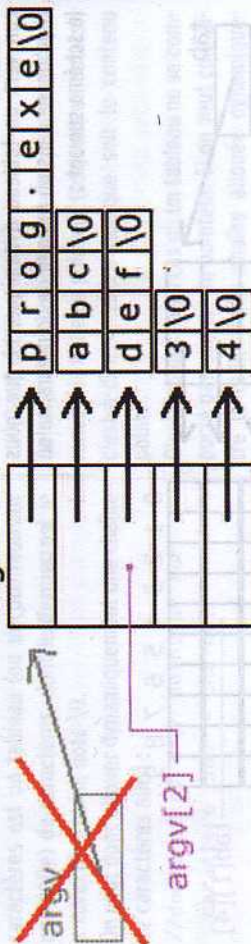
ou avec un tableau contenant la taille de chaque ligne (`int nb_colonnes[nb_lignes]`) :

```
for(i = 0; i < nb_lignes; i++)
    tab[i] = malloc(nb_colonnes[i]
        * sizeof(int));
```

La ligne de commandes

Lorsque vous donnez des paramètres à un programme via la ligne de commandes, ces paramètres vont en fait à la fonction `main()`, via les arguments `int argc` et `char * argv[]` : `argc` est le nombre d'arguments, et `argv` un tableau de pointeurs sur caractères (cf schéma), et non un pointeur sur pointeur sur caractères. Le nom du programme étant considéré comme un argument, si on appelle le programme ainsi :

argv



```
prog.exe abc def 3 4
```

On a :

```
argc = 5;
argv[0] = "prog.exe";
argv[1] = "abc";
argv[2] = "def";
argv[3] = "3";
argv[4] = "4";
```

Remarque : lancer un programme avec des arguments se fait dans un shell (Unix) ou dans une fenêtre MS-DOS (Windows). Il faut soit être dans le répertoire du programme (comme dans l'exemple) et on donne juste le nom du programme, soit donner son chemin d'accès, suivi des éventuels arguments.

Exemple : C:\MesProg\prog.exe abc def 3 4

Exercice : Faire un programme capable d'afficher les arguments passés à la ligne de commandes et le nombre de lettres de chacun.

ekv

Merci à Clad, Olivier L. et E. D. pour leurs précisions.

déclarée (ci-dessus, riri) ou allouée dynamiquement (cf ci-dessous).

Exemple :

```
unCanard = (struct canard *)
malloc(sizeof(struct canard));
unCanard->age = 12;
```

Dans le cadre de l'utilisation d'un pointeur, faite attention à la priorité des opérateurs. Le point est prioritaire sur l'étoile, d'où la nécessité des parenthèses :

```
(*unCanard).age = 12; //correct
*unCanard.age = 12; //incorrect
```

Les membres de la structure peuvent être des variables de n'importe quel type, des pointeurs, mais aussi des instances d'autres structures.

Exercice 1 : Faire un programme qui effectue, dans des procédures, la saisie et l'affichage des données d'une structure contenant au minimum un entier, une chaîne de caractères et une instance d'une autre structure.

```
#include <stdio.h>
```

```
//definitions
struct date {
    int jour, mois, annee;
};
```

```
struct canard {
    struct date naissance;
    char nom[30];
    int numero_cb;
};
```

```
//procedure
void saisie(struct canard
*ptr_Canard) {
    printf("Nom : ");
    scanf("%s", ptr_Canard->nom);
    printf("Date de naissance");
    printf("(jj mm aaaa) : ");
```

```
// ptr_Canard est un pointeur, mais
ptr_Canard->naissance n'en est
pas un.
```

```
scanf("%2d %2d %4d",
    &(ptr_Canard->naissance.jour),
    &(ptr_Canard->naissance.mois),
    &(ptr_Canard->naissance.annee));
// les parenthèses ci-dessus ne sont
pas obligatoires compte tenu de
la priorité des opérateurs
printf("No de carte bancaire : ");
scanf("%d", &ptr_Canard->numero_cb);
printf("Merci.\n\n");
}
```

```
//main
int main() {
    //declaration
    struct canard monCanard;
    //saisie des données
    saisie(&monCanard);
    //attention : monCanard n'est pas
    un pointeur
    printf("Hahaha, votre canard"
        "s'appelle %s ! Amusant !\n\n",
        monCanard.nom);
    //Affichage
    affichage(&monCanard);
    return 0;
}
```

Il vous reste à faire la procédure d'affichage, sur le modèle de saisie.

Comme vous l'avez sans doute remarqué, il est possible d'enchaîner les opérateurs « . » et « > ». Le sens de lecture est alors de droite à gauche, du contenu vers le contenant : dans a-b>c.d d est contenu dans c, lui-même contenu dans b, qui est un membre de a (attention : a et b sont des pointeurs sur structure, c est une structure et d une variable membre).

La déclaration typedef

Afin d'éviter de répéter à chaque fois « struct canard » ou « struct canard * », nous allons définir un nouveau type grâce à la déclaration typedef. Sa syntaxe est la suivante :

```
typedef description du type
nom_1_du_type;
et pour définir un nouveau type de pointeur :
typedef description du type
*nom_du_pointeur_sur_type;
```

On peut poser plusieurs noms en les séparant par des virgules. Cela permet notamment de définir en même temps le type et le pointeur sur ce type :

```
typedef      description du type
nom_1_du_type, *nom_du_pointeur_sur_type;
```

Exemple :

```
typedef unsigned long int entier_long;
//nous pouvons maintenant utiliser
entier_long a la place de unsigned
long int;
entier_long nb=100;
```

Dans le cas des structures, il est extrêmement courant d'utiliser typedef lors de la définition de la structure, car ces deux étapes peuvent se faire en même temps :

```
typedef struct date {
    int jour, mois, annee;
} date_t;

typedef struct canard {
    date_t naissance;
    char nom[30];
    int numero_cb;
} canard_t, *canard_p;
```

On crée ci-dessus un type « structure canard » (canard_t) et un type « pointeur sur structure canard » (canard_p). Ainsi, pour créer une ins-

tance de la structure canard, nous pouvons procéder de la manière suivante :

```
canard_t unCanard;
ou encore ainsi :
canard_p unCanard;
malloc(sizeof(canard_t));
```

De même, dans l'exercice 1 :

```
struct canard monCanard; et struct canard
*ptr_Canard peuvent être remplacés par :
canard_t monCanard; et canard_p
ptr_Canard
```

Notez qu'un type peut être défini avant la définition de la structure elle-même :

```
typedef struct date_t;
struct date {
    int jour, mois, annee;
};
```

Les tableaux de structures

À l'instar des variables classiques, les structures peuvent être regroupées par tableaux. Il suffit de rajouter, entre crochets, le nombre de structures que l'on désire avoir :

```
canard_t mesCanards[10]; //tableau
de 10 structures canard
```

Les tableaux de structures s'utilisent de la même manière que les tableaux « normaux » : mesCanards[3].numero_cb = 1737289;

Exercice : Écrire une fonction qui permette d'afficher les éléments d'un tableau de structure « canard » dont l'année de naissance est identique à celle entrée par l'utilisateur.

Je vous conseille de faire d'autres exercices avec les structures et les typedef avant de passer aux listes chaînées.

ek

LES LISTES CHÂÎNÉES

Lors que l'on manipule un nombre variable d'instances d'une structure, et que l'on souhaite pouvoir en insérer et supprimer dynamiquement, les tableaux de structures ne suffisent plus. Les listes chaînées proposent une autre approche légèrement plus complexe, mais puissante et indispensable.

Kézako ?

Une liste chaînée est un ensemble de structures X dont une des variables membres est de type « *pointeur vers X* ». Ainsi, chaque élément de la liste contient un pointeur vers l'élément suivant :

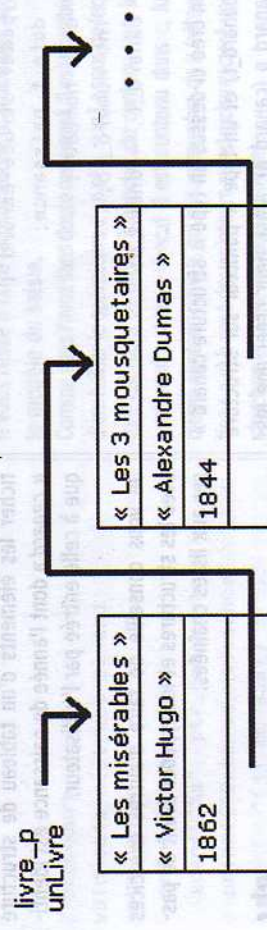
Exemple :

```
typedef struct livre {
    char titre[20];
    char auteur[30];
    int annee_parution;
    //pointeur vers le livre suivant
    struct livre *suivant;
} livre_t, *livre_p;

livre_t livre1, livre2, ...;
livre_n;
livre_p unlivre;
```

```
unlivre = &livre1;
//on passe au suivant
```

Une autre méthode, à mon avis plus efficace,



consiste à garder bien au chaud dans une variable le nombre d'éléments de la liste, et d'utiliser une boucle for :

```
//a mettre dans les declarations
int nb_elt = N;

//boucle parcourant tous les éléments
for(i=0; i<nb_elt; i++) {
    //operations sur unlivre ...
    unlivre = unlivre->suivant;
}
```

De plus, cela n'oblige pas le dernier pointeur à prendre une valeur particulière, ce qui nous permet de le faire pointer vers le premier élément. On obtient ainsi une liste circulaire, qui peut être traitée indifféremment avec ou sans début ni fin. Il faudra cependant faire attention à maintenir le nombre d'éléments à jour lors des ajouts et des suppressions d'éléments.

Liste doublement chaînée

Si, en plus d'un pointeur vers l'élément suivant, nous ajoutons dans les variables membres un pointeur vers l'élément précédent ? La liste est alors doublement chaînée : nous pouvons maintenant parcourir la liste dans les deux sens.

```
typedef struct livre {
    char titre[20];
    char auteur[30];
    int annee_parution;
    //pointeur vers le livre suivant
    struct livre *suivant;
    //pointeur vers le livre precedent
    struct livre *precedent;
} livre_t, *livre_p;
```

Exercice 1 : Écrire une fonction permettant d'insérer un élément dans la liste de livres définie ci-dessus.

Elle reçoit un pointeur sur un des livres, doit insérer le nouveau livre avant celui-ci, et retourner un pointeur vers le livre inséré. On incrémentera la variable globale nb_livre (de type unsigned int).

```
livre_p ajouter_livre(livre_p courant)
{
    livre_p nouveau;

    //Allocation de l'espace memoire
    nouveau = (livre_p) malloc(sizeof(livre_t));

    //Proprietes du livre
    printf("Titre : ");
    scanf("%s", nouveau->titre);
    printf("Auteur : ");
    scanf("%s", nouveau->auteur);
    printf("Annee de parution : ");
    scanf("%s", nouveau->annee_parution);

    //Actualisation des liens de la
    //liste chainee (voir schema)
    nouveau->suivant = courant; // (1)
    nouveau->precedent =
        courant->prec; // (2)
    courant->prec = nouveau; // (3)
    nouveau->precedent->suivant =
        nouveau; // (4)
}
```

```
//Actualisation du nombre d'objets
nb_obj ++;

return nouveau;
}
```

Je vous laisse le soin de faire la fonction (ou procédure, au choix) retirer_livre(livre_p livre_a_retirer).

FONCTIONS RÉCURSIVES

La notion de récursivité est très simple et assez utile pour un bon nombre d'algorithmes. Elle ne s'applique pas qu'aux fonctions, mais celles-ci en sont l'exemple le plus concret et didactique, donc celui que nous allons voir :

Définition

Une fonction récursive est une fonction qui s'appelle elle-même, ni plus, ni moins.

Exemple : la factorielle d'un entier positif n . On rappelle que la factorielle d'un nombre entier est le produit de tous les entiers inférieurs ou égaux à ce nombre : la factorielle de 5 est $1 \times 2 \times 3 \times 4 \times 5$ et par convention, la factorielle de 0 vaut 1.

```
long int fact(unsigned int n) {
    if(n==0) return 1;
    else return (n*fact(n-1));
}
```

Faisons tourner la fonction « à la main », en lui passant 3 comme argument :

```
fact(3) = 3 * fact(2)
        = 3 * 2 * fact(1)
        = 3 * 2 * 1 * fact(0)
        = 3 * 2 * 1 * 1
```

On a donc bien la factorielle de 3.

La fonction s'est appelée elle-même en décrémentant à chaque fois son argument. La fonction en cours se met en pause et génère un clone dont elle attend le résultat. Ce clone s'exécute et en génère un autre, et ainsi de suite.

Condition d'arrêt

Évidemment, si l'on laissait la fonction s'appeler indéfiniment, cela poserait un petit problème. C'est pourquoi il est nécessaire que la récursivité soit toujours conditionnée par un test déterminant si elle doit se poursuivre ou non : la condition d'arrêt. Dans notre exemple, il s'agit du test $\text{if}(n=0)$: en effet, l'argument se décrémentant à chaque fois, on regarde s'il n'a pas atteint 0, auquel cas on

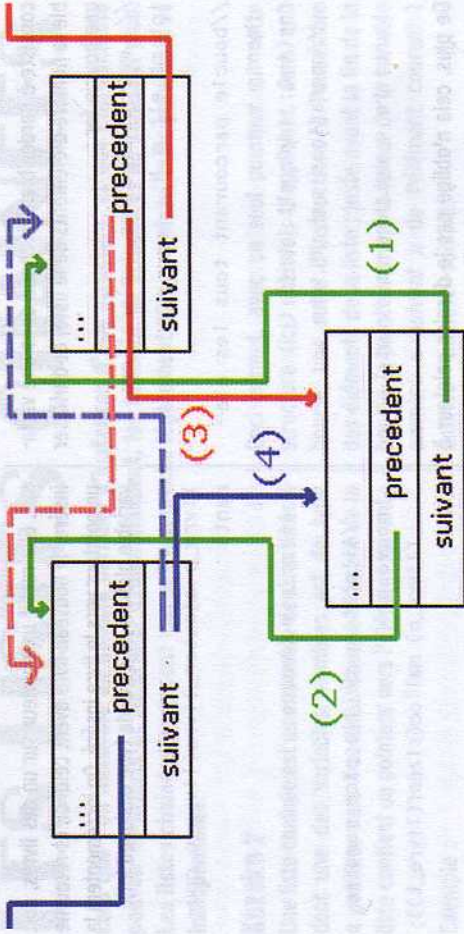
arrête la récursivité. La fonction retourne alors 1 au clone qui l'avait appelé, lequel retourne le produit de cette valeur avec son propre argument, et ainsi de suite. On remonte donc des appels vers les appelants, comme des assiettes empilées qu'on dépile, jusqu'à la fonction initiale, qui retourne la valeur de la factorielle.

Exercice : écrire une fonction récursive qui permette d'élever un nombre a à la puissance b .

Problème des fonctions récursives

Lorsqu'une fonction (appelant) se met en attente pour en appeler une autre (appelée), l'état d'exécution de l'appelant ainsi que toutes ses variables sont stockés dans un espace de la mémoire vive : la pile d'appels. Ceci permet de reprendre l'exécution de l'appelant au bon endroit, et de retrouver ses variables. De même, une fonction récursive sauvegarde ses données dans la pile. Donc, si la récursivité est trop importante, les assiettes dépassent la taille du placard, on obtient un dépassement de pile (ou explosion de pile). Le nombre d'appels autorisés dépend du compilateur, de l'OS et de la machine, mais reste généralement suffisant pour un traitement moyennement important ; la limite commence à se faire sentir lorsqu'on traite une quantité importante de données par une récursivité. Par exemple, traiter dans une image, en partant d'un pixel, tous les pixels autour et ce, de manière récursive, conduira à coup sûr à un débordement de pile, sauf si l'image est très petite. Dans ces cas, on utilisera plutôt un algorithme itératif, c'est-à-dire une boucle, avec un `while` ou un `for`.

ek



Structures imbriquées

Comme nous l'avons vu dans le cours sur les structures, celles-ci peuvent être imbriquées, c'est-à-dire qu'une structure peut, dans ses variables membres, contenir une autre structure. Il en est de même pour les listes chaînées.

Exemple :

```
typedef struct point {
    int x, y; //position du point
    struct point *suiv;
    struct point *prec;
    point_t, *point_p;
}
```

```
typedef struct forme {
    int x, y; //position du centre
    int nb_pts; //nombre de points
    point_p unPoint;
    forme_t, *forme_p;
}
```

La structure `forme` contient une liste chaînée `point`, et est elle-même une liste chaînée. Si on veut traiter chaque point de chaque forme, une double boucle `for` suffit (on a déjà le nombre de points dans chaque forme, il nous faut juste le nombre de formes).

Thanks : Olivier L., Clad & dvrasp

ek

GDB, MAÎTRE DU DÉTOURNEMENT

Lorsque l'origine d'un boque tarde à être découverte, lorsque des dizaines de processus tournent sur notre système sans que l'on puisse apparemment en contrôler le flux ou en lire la mémoire, une solution utilisateur s'impose toujours d'elle-même : Gnu DeBugger.

GDB est un outil d'analyse conçu principalement pour aider les développeurs à résoudre des problèmes techniques en leur fournissant un plan de travail bas-niveau basé sur l'appel système `ptrace()`. Ainsi GDB permet de contrôler le flux d'exécution d'un programme, de l'interrompre arbitrairement, de lire et modifier sa mémoire, ses registres. Avec sa capacité à désassembler du code à la volée, GDB sera aussi d'une aide conséquente dans des travaux de reverse engineering ou d'autopsies.

GDB comprend les symboles d'un binaire ELF et fournit ainsi un suivi d'exécution clair : lorsqu'un programme est arrêté (signal, erreur) il est possible de remonter les noms des fonctions qui ont précédé son arrêt grâce à un suivi des `stack-frames`.

Si les codes sources du logiciel sont disponibles et que le binaire a été compilé pour pouvoir être débogué avec GDB, un suivi d'exécution avec le code source rendra les travaux de débogage beaucoup plus simples.

Pour vos premiers essais, écrivez le fichier suivant `toto.c` :

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
```

```
int main(int ac, char **av) {
    int i;
    i = atoi(av[ac-1]);
    if (i == 0) {
        printf("Oui, i est egal a
               [0]\n");
        return (0);
    }
    printf("Non, i est egal a [%d]\n",
           i);
    while (1) {
        sleep(5);
    }
    return (0);
}
```

Ce programme prend le dernier argument transmis, un chiffre, et l'affecte à la variable `i`. Si le paramètre est « 0 », alors le programme confirme que `i` est égal à 0. Sinon il affiche sa valeur. Le hic c'est que ce programme ne s'arrête jamais si `i` est différent de 0 ; il ne consomme néanmoins, grâce à `sleep()`, que très peu de ressources. On le compilera avec la ligne de commandes suivante :

```
$ gcc -W -Wall -ggdb -o toto toto.c
```

Les options `-W` et `-Wall` spécifient un haut niveau d'avertissement, chose utile si votre code contient un oubli qui n'empêche pas sa compila-



tion. L'option `-ggdb` demandera au compilateur d'intégrer des symboles de débogage. Cette option a la particularité de demander l'utilisation du format d'informations le plus complet.

Naviguer dans les `stack-frames`

Lancez maintenant GDB. Chargez votre exécutable en y tapant :

```
(gdb) file toto
```

GDB va y lire les symboles, c'est à dire qu'il connaîtra les noms des fonctions et les adresses qui leurs sont associées. Lancez le programme à l'aide de la commande « `run` » qui a un raccourci « `r` ». Le paramètre du programme sera « 2 » :

```
(gdb) r 2
```

Le programme se lance et boucle. Faites `[ctrl + z]` pour stopper le processus avec un signal SIGSTP. GDB vous rend alors la main en vous indiquant à quelle adresse s'est arrêtée l'exécution.

À ce moment, si l'on relance le processus, il continuera là où il s'est arrêté, c'est à dire endormi dans l'exécution de la boucle. Tapez la commande « `backtrace` » qui a un alias « `bt` » :

```
(gdb) bt
#0 0x400c6aa5 in nanosleep() from
/lib/libc.so.6
#1 0x400c690f in sleep() from
/lib/libc.so.6
#2 0x08048427 in main (ac=2,
av=0xbffff74) at toto.c:16
```

Les trois lignes qui apparaissent sont un suivi des appels qui ont eu lieu pour arriver au point d'exécution actuel. Les adresses en début de ligne désignent l'endroit où doit se poursuivre l'exécution au retour de la fonction ou de l'appel système. On sait donc que le programme est parti de `main()` pour arriver jusqu'à la fonction d'interface `sleep()` - en l'appelant à la ligne 16 du

fichier toto.c. Dans la lib, sleep() appelle ensuite nanosleep().

On peut utiliser la commande « list » pour afficher une portion du code source disponible autour de la ligne 16 :

```
(gdb) list toto.c :16
```

Mais comment ferions-nous si nous voulions, par exemple, vérifier quelle valeur a actuellement la variable i dans notre fonction main() ? Il nous faudrait disposer de la région de la pile (stack frame) qui lui est associée. Pour cela on utilisera la commande « select-frame » qui a un raccourci « f » :

```
(gdb) f 2
```

De là on pourra accéder aux variables locales de la fonction main() (associée à l'index 2 comme vu plus haut). La commande « whatis » nous permettra de vérifier le type de la variable. La commande « print », dont le raccourci est « p », nous révélera sa valeur actuelle :

```
(gdb) whatis i
```

```
type = int
```

```
(gdb)p i
```

```
$1 = 2
```

```
(gdb)p $1
```

```
$2 = 2
```

Vous aurez noté que print affecte automatiquement le résultat à une variable numérique. L'avantage est que vous pourrez réafficher la valeur de la variable i sans avoir à vous situer sur la région mémoire associée à main(). On aurait pu utiliser la commande « printf » qui ne fait pas d'affectation :

```
(gdb) printf "i = %d\n", i
```

```
i = 2
```

Vous savez désormais vous déplacer dans une stack-frame. Poursuivons en comprenant comment manipuler le flux d'exécution du programme.

Flux d'exécution : breakpoints, watchpoints et désassemblage

Chargeons à nouveau notre exécutable toto (« file toto »). Cela aura pour effet de tuer l'exécution actuelle. À ce stade, le programme n'est pas lancé. Vous pouvez néanmoins en désassembler le code. Comme nous avons les codes sources pour la fonction main, nous ne nous privons pas de les utiliser pour nous simplifier la tâche. Tapons donc :

```
(gdb) list main
```

Et voilà que défile le code associé à la fonction main. Nous aimerions arrêter l'exécution juste avant que l'affectation de la valeur de retour d'atoi() ne soit réalisée. Dans notre exemple, le code associé est à la ligne 9 : « i = atoi(argv[1]) ». La commande info va nous permettre d'obtenir les adresses du code exécutable associées à l'exécution de cette instruction C.

```
(gdb) info line 9
```

```
Line 9 of « toto.c » starts at address 0x80483d4
<main+16> and ends at 0x80483ed <main+41>
```

On apprend alors que le code exécutable pour réaliser l'appel de fonction puis l'affectation est localisé entre les adresses 0x80483d4 (d'offset 16 par rapport au début de la fonction main) et 0x80483d4(d'offset 41). Le code de cette zone est donc de 25 octets. On peut désassembler arbitrairement des régions exécutables de la mémoire, ce qui nous permettra d'obtenir strictement le code qui nous intéresse :

```
(gdb) disass 0x80483d4 0x80483ed
```

```
0x08483e5 <main+33>:call 0x08482f0
```

```
<atoi>
```

```
0x80483ea <main+38>:mov %eax, %ecx
```

```
0xfffffff0 <main+39>:mov %ecx, %eax
```

```
0xfffffff0 <main+40>:mov %eax, %ecx
```

```
0xfffffff0 <main+41>:mov %ecx, %eax
```

Notez que l'on aurait pu saisir « disass main » qui entraîne le désassemblage de toute la fonction main.

tion main(). C'est utile lorsque l'on ne dispose pas des sources, si l'on voulait désassembler nanosleep() de la lib par exemple.

La dernière ligne du code en assembleur correspond à l'affectation de la valeur de retour d'atoi(), contenue dans le registre EAX. Si l'on place un point d'arrêt sur l'adresse 0x80483ea, alors l'exécution s'arrêtera exactement avant la réalisation de l'instruction mov.

La commande « break » permet de placer des breakpoints. On utilisera son raccourci « b » :

```
(gdb) b *0x80483ea
```

Breakpoint 1 at 0x80483ea: file toto.c, line 9.

Nous allons maintenant lancer le programme, en tapant « r 2 », comme précédemment. Très rapidement l'exécution s'arrête, et l'on voit que c'est du fait d'un breakpoint. Une demande d'information sur les registres EAX et EIP nous confirmerait que EAX est bien égal à deux (notre argument), et qu'EIP pointe sur 0x80483ea.

```
(gdb) info reg eax eip
```

```
eax 0x2
```

```
eip 0x80483ea 0x80483ea
```

Bien, nous allons donc maintenant demander à GDB de s'arrêter dès que la valeur de la variable i aura changé. Ça ne devrait pas être très long puisque nous sommes justement sur le point d'exécuter l'instruction qui réalisera cette action. Mais nous sommes paranoïaques et nous voulons être sûrs que l'affectation aura bien lieu !

On utilisera la commande « watch » qui placera un watchpoint sur une expression. On effacera le breakpoint actuel à l'aide de « delete » (raccourci par « d » suivi du numéro du breakpoint) et on relancera le programme dans son exécution avec « next », qui permet d'exécuter les instructions assembleur une par une :

```
(gdb) watch i
```

```
Hardware watchpoint 2: i
```

```
(gdb) d 1
```

```
(gdb)nexti
```

```
Hardware watchpoint 2: i
```

```
Old value = -1073742732
```

```
New value = 2
```

```
(à)
```

Les lignes non recopiées concernent le point d'arrêt de l'exécution. On s'est arrêté juste après l'affectation de la valeur de i par atoi(). Peut-être peut-on changer la valeur de i en mémoire et dupier la vérification qui suit ? C'est certain, et notez que l'on aurait aussi pu agir, immédiatement après le retour d'atoi(), sur le registre EAX.

On efface notre point d'arrêt devenu inutile, et on modifie la mémoire avec « set » :

```
(gdb) d 2
```

```
(gdb) var i=0
```

```
(gdb) cont
```

```
Continuing.
```

```
Oui, i est égal à [0]
```

Program exited normally.

Facile. On aurait aussi pu modifier la valeur de i si nous avions son adresse :

```
(gdb) p &i
```

```
$2 = (int *) 0xbffffc24
```

```
(gdb)set var * 0xbffffc24=0
```

Entraînement cérébral : exercez-vous !

Au terme de cet article, vous devriez être capable de réaliser l'exercice suivant : lancez le programme toto dans un shell avec un argument différent de « 0 », et dans un autre attachez le processus à GDB avec la commande :

```
$ gdb ./toto PID_TOTO
```

Votre objectif est de faire sortir toto de la boucle infinie qui l'empêche de quitter correctement. Alors, c'est l'histoire de Toto qui n'arrivait pas à utiliser GDB...

Clod Strife

FAITES POUSSER UN ARBRE... BINAIRE !

RAPPELS

Afin de comprendre la notion d'arbres binaires ainsi que son utilité, il est nécessaire de rappeler les bases d'une structure de données plus simple qui est la liste chaînée (cf. page 32, "Les listes chaînées").

La liste chaînée est composée d'une suite de maillons qui contiennent une donnée et un « lien » vers le maillon suivant. En ce qui concerne le dernier maillon, il n'y a plus rien à atteindre ensuite, donc le lien n'a pas de raison d'exister. En C, ce lien aura la valeur NULL (définie dans le header stdlib.h qui permettra donc de déterminer la fin de la liste chaînée) ; mais nous reviendrons sur ce point après.

Le schéma suivant représente ce que nous venons d'expliquer :



Fig 1. Liste chaînée

En outre, il est tout à fait possible de créer en plus un lien du maillon courant vers le maillon précédent. La liste est appelée liste doublement chaînée dans ce cas. Écrivons la structure d'un des maillons de la chaîne en C en considérant que chaque maillon contient une valeur entière :

```

struct s_maillon {
    int valeur;
    /* lien vers le prochain maillon */
    struct s_maillon *prochain;
    /* lien vers le precedent */

```

```

    struct s_maillon *precedent;
};

```

Après ces rappels, nous pouvons maintenant aborder la notion d'arbre binaire.

Caractéristiques des arbres binaires

DÉFINITION D'ARBRE BINAIRE

Visualisons tout d'abord un arbre binaire de manière générale et intuitive :

On remarque que chaque noeud de l'arbre a deux fils (qui sont des sous-arbres) que l'on appellera fils gauche pour le sous-arbre gauche

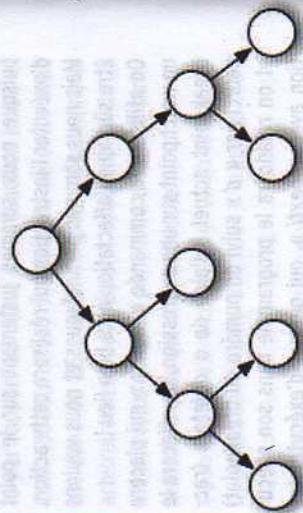


Fig 2. Arbre binaire

et fils droit pour le sous-arbre droit. Un noeud peut avoir seulement un fils gauche ou un fils

Le code suivant montre comment créer l'arbre de la figure précédente. On remarquera que la fonction `add_node_in_tree` insère l'élément de façon ordonnée, suivant le pseudo-algorithme :

```

#include <stdio.h>
#include <stdlib.h>

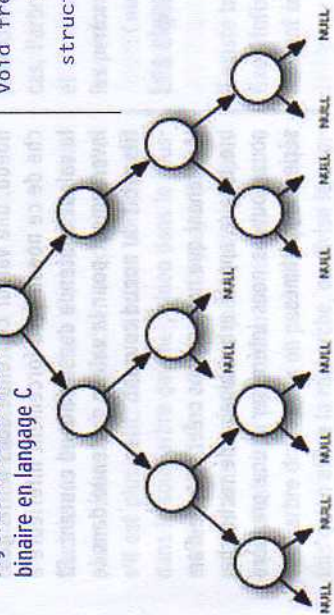
struct s_node {
    int valeur;
    struct s_node *fg; /* fils gauche */
    struct s_node *fd; /* fils droite */
};

struct s_node *create_node(
    int valeur) {
    struct s_node *tmp;
    if (!(tmp =
        malloc(sizeof(struct s_node))))
        exit(1);
    /* allocation impossible */
    tmp->valeur = valeur;
    tmp->fg = NULL;
    tmp->fd = NULL;
    return tmp;
}

void free_tree(
    struct s_node **tree) {
    struct s_node *tmp = *tree;
    if (tmp->fg)
        free_tree(&(tmp->fg));
    // on appelle la libération
    // mémoire sur le
    // fils gauche
    if (tmp->fd)
        free_tree(&(tmp->fd));
}

```

Fig 3. Zoom sur un arbre binaire en langage C



```

// libération mémoire sur le
// fils droit
free(*tree); // libération du
// noeud courant
*tree = NULL ;
}

void add_node_in_tree(
    struct s_node *tree,
    int value) {
    struct s_node *tmp = tree;
    /* initialisation sur la racine
    du sous-arbre en cours */
    if (value < tmp->value) {
        if (tmp->fg == NULL)
            tmp->fg = create_node(value);
        else
            add_node_in_tree(tmp->fg,
                value);
    }
    if (value > tmp->value) {
        if (tmp->fd == NULL)
            tmp->fd = create_node(value);
        else
            add_node_in_tree(tmp->fd,
                value);
    }
}

void add_node(struct s_node **tree,
    int value) {
    if (*tree == NULL)
        /* l'arbre n'existe pas encore */
        *tree = create_node(value);
    /* creation de la racine de
    l'arbre */
    else
        add_node_in_tree(*tree,
            value);
}

```

TRADUCTION PSEUDO-CODE :

Si la valeur à insérer est inférieure à la valeur du noeud courant

Si le fils gauche est à NULL alors insérer la valeur dans un noeud fils gauche

Sinon insérer la valeur dans le sous-arbre gauche

Sinon si la valeur du fils droit est supérieure à la valeur du noeud courant

Si le fils droit est à NULL alors insérer la valeur dans un noeud fils droit

Sinon insérer la valeur dans le sous-arbre droit

Sinon ne rien faire pour valeur égale à la valeur du noeud courant

bre de noeuds où l'on peut trouver l'élément par deux.

Algorithmes sur les arbres binaires de recherche

LA RECHERCHE

Pour rechercher un élément dans un arbre binaire de recherche, on compare cet élément au contenu de la racine :

En cas d'égalité, l'élément a été trouvé et on arrête la recherche.

Si l'élément est inférieur (respectivement plus grand) à la racine, on relance la recherche sur le sous-arbre gauche (respectivement sous-arbre droit). Si ce sous-arbre est NULL, alors l'élément n'a pas été trouvé et la recherche a échoué.

On peut se rendre compte de la puissance des arbres binaires par rapport aux autres structures de données plus simples. Il suffit de réfléchir à la recherche d'un élément dans le pire cas pour les différents types de structures de données : d'un côté la liste, et de l'autre, l'arbre binaire de recherche.

Dans le cas de la structure d'arbre, on va comparer l'élément à rechercher avec la racine et s'il est plus petit, on va effectuer la recherche dans le sous-arbre gauche, sinon dans le sous-arbre droit. Si l'on admet que la proportion de noeuds dans le sous-arbre gauche est à peu près équivalente à la proportion de noeuds dans le sous-arbre droit, alors on peut dire qu'à chaque fois qu'on affine la recherche, on divise le nom-

Alors, dans le pire des cas on a $650/20^m$ itérations avant de trouver l'élément. Il faut trouver m qui correspond à la première fois que la fraction précédente va être inférieure à 1. En utilisant le logarithme népérien, on trouve $n = 16$.

Grâce à la structure d'arbre binaire de recherche, on a pu considérablement diminuer le nombre d'itérations dans le cas d'une recherche d'éléments par rapport à une liste chaînée. Le gain de temps est lui aussi considérable, d'autant que les éléments présents peuvent être beaucoup plus nombreux et la comparaison des éléments plus longue que de comparer un int. Par exemple, on pourrait construire un arbre correspondant à tous les mots du dictionnaire où la recherche dépendrait de l'ordre lexicographique et donc d'un appel à la fonction strcmp pour chaque comparaison d'élément. Dans ce cas, l'arbre binaire de recherche est inévitable et s'avère être un très bon moyen d'améliorer les performances d'un programme.

Les algorithmes de parcours

Il existe trois types de parcours possibles : préfixe, infixe, postfixe qui correspondent au moment où le noeud va être traité.

Le troisième paramètre (mode) est optionnel, il n'est utile que dans le cas où le fichier est créé par le programme.

Ce paramètre prend la même valeur que celle que vous donneriez à la commande `chmod`. Par exemple, `open("my_file", O_RDWR, 0644)`; ouvrira un fichier nommé `my_file` en lecture / écriture avec les droits 644 en octal.

Petit rappel :

```
sh> chmod 644 my_file
sh> ls -l
-rw-r--r-- 1 login group 1407
Mar 1 13:52 my_file
sh>
```

Les flags servent quant à eux à indiquer au système les opérations qui seront effectuées sur le fichier. Par exemple, `O_RDONLY` ouvrira le fichier en lecture seule, `O_WRONLY` en écriture seulement et `O_RDWR` ouvrira le fichier en lecture/écriture.

D'autres flags peuvent être utilisés pour, par exemple, indiquer au système que toutes les écritures devront se faire en fin de fichier (`O_APPEND`) ou encore, dans le cas où le fichier n'existe pas, il sera créé si le flag `O_APPEND` est spécifié.

Si vous désirez en savoir plus sur les flags, leur utilisation est clairement spécifiée dans le manuel de « `open` ». Je vous invite donc à taper « `man 2 open` » dans votre shell favori pour la liste exhaustive des flags et leurs utilisations.

Enfin, l'appel système « `open` » nous retourne un int correspondant à un descripteur de fichier (file descriptor). « `open` » renvoie toujours le plus petit file descriptor libre.

En cas d'échec, « `open` » renvoie -1. Il est donc impératif de vérifier la valeur retournée par « `open` ».

```
if ((fd = open("my_file",
O_RDONLY)) == -1) {
    perror("open");
    exit(0);
}
```

On referme un fichier avec la fonction `close` : `int close(int fd)`; Cette fonction referme le fichier dont le descripteur est `fd`. En cas de réussite, elle retourne 0, sinon elle retourne -1. Il est également indispensable de TOUJOURS penser à fermer le descripteur qui a été ouvert avec « `open` ».

Lecture d'un fichier

Une fois le fichier ouvert, il devient possible de commencer à le lire.

C'est l'appel système « `read` » qui sert d'interface au développeur pour la lecture d'un fichier. Son prototype est : `ssize_t read(int d, void *buf, size_t nbytes)`;

« `read` » prend en paramètre le file descriptor du fichier à lire (le même que celui retourné par « `open` »), un pointeur sur la zone mémoire préalablement allouée en vue de la récupération des données et le nombre d'octets à lire. La valeur retournée par « `read` » correspond au nombre d'octets effectivement lus. « `read` » renvoie zéro lorsqu'il arrive à la fin du fichier. Tout comme « `open` », « `read` » retourne -1 en cas d'échec de la lecture.

Maintenant que nous maîtrisons la lecture et l'écriture d'un fichier, nous allons pouvoir réaliser notre premier programme qui va ouvrir un fichier et afficher son contenu à l'écran (une sorte de mini commande `cat`).

```
sh> cat minicat.c
int main(int argc, char
*argv[])
```

```
{
    char c;
    int fd;

    if (argc == 2) {
        if ((fd = open(argv[1],
O_RDONLY)) == -1) {
            perror("open");
            exit(0);
        }
        while (read(fd, &c, 1) > 0)
            putchar(c);
        close (fd);
    } else {
        fprintf(stderr, "Usage: %s
'file'\n", argv[0]);
        return (-1);
    }
    return (0);
}
```

```
sh> gcc -o minicat minicat.c
```

```
sh> ./minicat minicat.c
```

```
...
sh>
```

Positionnement dans un fichier

Il peut souvent être utile de pouvoir se positionner où on le veut dans le fichier. La fonction `lseek()` nous permet cette opération.

`off_t lseek(int fd, off_t offs, int whence);`

`lseek` repositionne l'offset du file descriptor `fd` à l'offset `offs` selon la directive `whence`.

Whence est un flag qui peut prendre trois valeurs : Si whence vaut `SEEK_SET`, l'offset est positionné à `offs` octets.

Si whence vaut `SEEK_CUR`, l'offset est positionné à l'offset courant plus `offs` octets.

Si whence vaut `SEEK_END`, l'offset est positionné à la taille du fichier plus `offs` octets.

En cas d'échec, `lseek` retourne -1. L'utilisation de `lseek` se révèle particulièrement utile dans le cas de gros fichiers, permettant ainsi de ne pas parcourir l'intégralité du fichier pour écrire à l'endroit souhaité.

Écriture dans un fichier

Il est possible d'écrire dans un fichier (et de manière plus générale, dans un descripteur) avec l'appel système « `write` » :

`ssize_t write(int d, const void *buf, size_t nbytes)`; « `write` » prend donc en paramètre le file descriptor dans lequel il doit écrire, un pointeur sur la zone mémoire à écrire et la taille du buffer à écrire. Tout comme son ami « `read` », « `write` » retourne le nombre d'octets écrits et -1 en cas d'erreur.

Il nous suffirait donc, dans notre `minicat.c`, de remplacer la ligne « `putchar(c)` » par « `write(1, &c, 1)` » pour obtenir le même résultat.

Un peu d'optimisation

Dans notre exemple (`minicat`), nous faisons appel à « `read` » et « `write` » à chaque itération de boucle.

En effet, nous lisons les caractères un à un et les écrivons aussitôt sur la sortie standard (nous aurions pu tout aussi bien écrire dans un fichier en changeant le file descriptor passé à « `write` », et ainsi transformer notre `minicat` en `minicp`).

L'inconvénient de cette méthode est qu'elle est très coûteuse en temps. Le noyau a en effet besoin d'un certain temps pour « invoquer » un appel système. Même si cela n'a rien de vraiment significatif dans notre exemple de `minicat`, une bonne habitude à prendre et d'éviter de faire des appels système de manière excessive. Au lieu de lire/écrire un à un les caractères, une solution serait donc de bufferiser les données

afin de réduire le nombre d'appel de « read » et « write ». Pour cela, nous ferions quelque chose du genre :

```
buf = malloc(BUFLen);
/* fd est le file descriptor du
fichier à copier */
while (read(fd, buf, BUFLen)) {
    write(fd2, buf, BUFLen);
/* fd2 étant le fd du fichier de
destination */
```

Je vous invite à essayer de copier un fichier très volumineux avec votre minicp (> 10M), avec un BUFLen à 1 (il y a des chances pour que vous abandonniez avant la fin), puis avec un buffer de 100 ou 1000, vous aller trouver la différence plus qu'appréciable ;-).

Place à la pratique !

Maintenant que nous commençons à avoir de solides bases théoriques en ce qui concerne la manipulation de fichiers, nous allons pouvoir mettre tout cela en pratique. Nous allons donc nous appuyer sur un exemple concret : un contrôleur d'intégrité.

Ce petit programme aura pour but de stocker dans une base (un fichier) des sommes MD5 et de les vérifier, contrôlant ainsi si le fichier a subi une modification depuis sa dernière entrée en base.

Le MD5 est un algorithme de cryptage qui a été créé par le professeur Ronald L. Rivest du MIT, et qui prend un message d'une taille arbitraire et produit une clé de 128 bits à partir de ce message. Nous allons donc commencer dans un premier temps, comme pour tout programme, par vérifier la syntaxe de la ligne de commande. Ensuite, si l'option « -c » est passée au programme, nous vérifierons les sommes de tous les fichiers pas-

sés en paramètre, sinon, nous les écrirons dans la base.

```
#define BASE_NAME "database"
#define SYNTAX_ERROR ": [-c]
files..."

/* Fonction appelée en cas d'erreur
de syntaxe */
void usage(char *progname)
{
    /* progname est le nom du programme
    ** (passe en paramètre à la
    ** fonction) */
    fprintf(stderr, "%s %s\n",
        progname, SYNTAX_ERROR);
    /* On quitte le programme */
    exit(0);
}
```

```
int main(int argc, char *argv[])
{
    /* Tableau contenant les noms
    des fichiers passés en paramètre */
    char **filename;
```

```
/* Le nom de notre base */
char *database_name = BASE_NAME;
/* Flag égale 1 si l'option '-c'
** est spécifiée */
int check = 0;
/* Notre programme prend au
** moins un paramètre */
if (argc < 2) {
    usage(argv[0]);
}
/* Si l'option '-c' est passée
** au programme */
if (!strcmp(argv[1], "-c", 2)) {
```

```
/* arguments, sinon, on quitte
** le programme. */
if (argc < 3) {
    usage(argv[0]);
}
check = 1;
filename = &argv[2];
} else {
    filename = &argv[1];
}
/* On vérifie les signatures si
** '-c' est spécifiée */
if (check)
    check_sum_in_file(database_name,
        filename);
else {
    /* sinon on écrit les sommes
    ** dans la base */
    write_sum_in_file(database_name,
        filename);
}
return (0);
}
```

Écriture dans la base

Nous allons donc commencer par l'écriture des sommes MD5 des fichiers passés en paramètre dans la base. La fonction get_sum s'occupera de générer une somme MD5 à partir du fichier.

On passe en paramètre un tableau contenant le nom des fichiers, elle nous renvoie un tableau contenant les sommes associées à ces fichiers. Pour générer une somme à partir d'un fichier, il existe peut-être sous votre OS la fonction MD5File().

Cette fonction prend en paramètre le chemin du fichier à traiter et un buffer pour stocker le résultat. Nous stockons le résultat dans un tableau (char **digest), nous ne nous en préoccupons donc pas. La fonction MD5File nous renvoie NULL en cas d'échec de la génération de la somme.

Notre fonction get_sum retournera donc NULL également.

```
char **get_sum(char **filename)
{
    char **digest;
    int i;

    digest = malloc(sizeof(filename));
    for (i = 0; filename[i]; i++) {
        digest[i] =
            malloc(strlen(filename[i]));
        if ((digest[i] =
            MD5File(filename[i],
            NULL)) == NULL) {
            fprintf(stderr,
                "%s: No such file\n",
                filename[i]);
            return (NULL);
        }
        return (digest);
    }
}
```

Vous remarquerez l'utilisation de my_MD5File. MD5File étant une fonction qui existe sous BSD, mais pas sous Linux.

Étant donné qu'il y a plus de chance que nos chers lecteurs soient adeptes des pingouins plutôt que de mon diable adoré, voilà une petite fonction qui va générer une empreinte à partir d'un fichier. Vous noterez que, pour changer un peu, nous utilisons ici fopen() et fread().

```
char **my_MD5File(char *filename)
{
    FILE *fp;
    MD5_CTX context;
    int len;
    unsigned char buffer[1024];
```

```

unsigned char digest[16];
unsigned int i,j;
char *u;

if ((file = fopen(filename, "rb"))
    == NULL) {
    printf ("%s cannot be opened\n",
            filename);
    return (NULL);
}

MD5Init(&context);
while (len = fread(buffer, 1, 1024,
                    file))
    MD5Update(&context, buffer, len);
MD5Final(digest, &context);
u = malloc(16);
for (i = 0, j=0; i < 16; i++, j+=2)
    sprintf(&u[j], "%02x", digest[i]);
fclose (file);
return (strdup(u));
}

```

Notre fonction `write_sum_in_file` va s'occuper, comme son nom l'indique, d'écrire les sommes dans la base. Elle appelle donc dans un premier temps la fonction `get_sum()` et récupère les sommes MD5 associées aux fichiers dans `digest`. Ensuite nous ouvrons le fichier `database` avec les droits en écriture seulement (nous n'avons pas à lire le fichier, nous l'ouvrons donc avec les droits minimums).

Si le fichier n'existe pas, nous le créons et toutes les écritures se feront à la fin du fichier (si, si ! souvenez vous des flags `O_APPEND` et `O_CREAT`). Enfin, nous appliquons les droits 644 en octal à ce fichier (si vous souhaitez que la base ne puisse pas être consultée par un tiers, vous n'avez qu'à changer le 0644 par 0600 par exemple).

En cas d'échec de l'appel système « `open` », nous quittons notre fonction. Ensuite, nous par-

courons notre tableau de noms de fichiers en même temps que notre tableau contenant leurs sommes MD5.

Nous écrivons dans un buffer le nom de fichier associé à sa somme, ensuite, nous écrivons toute la ligne dans notre fichier. Enfin, avant de quitter notre programme, nous pensons à fermer le file descriptor et à libérer le buffer alloué.

```

void
write_sum_in_file(char *database_name,
                  char **filename)
{
    char **digest;
    char *buf;
    int fd, i;

    if ((digest = get_sum(filename))
        == NULL)
        return ;

    if ((fd = open(database_name,
                   O_WRONLY|O_APPEND|O_CREAT,
                   0644)) < 0) {
        perror("open");
        return ;
    }

    buf = malloc(BUFLen);
    for (i = 0; filename[i]; i++) {
        sprintf(buf, "%s=%s\n",
                filename[i], digest[i]);
        if (write(fd, buf, strlen(buf))
            < 0) {
            perror("write");
            return ;
        }
    }
    close(fd); free(buf);
}

```

Vérification de l'intégrité des fichiers

La fonction `check_sum_in_file` va générer les sommes MD5 des fichiers passés en paramètre s'ils existent, et les comparer à celles présentes dans la base.

Pour cela, après avoir récupéré dans un tableau les sommes, nous ouvrons la base en lecture seule afin de la lire ligne par ligne grâce à la fonction `getline`, que je vous laisse développer à votre guise. Cette fonction prend en paramètre un file descriptor et retourne une ligne après l'autre (pensez tout de même à bufferiser et non lire les caractères un à un).

Ensuite, nous découpons cette ligne grâce à la fonction `str_wordtab` (que je vous laisse également développer tout seul comme un grand), qui prend en paramètre une chaîne et un délimiteur et nous renvoie un tableau de la chaîne coupée au délimiteur. Dans notre exemple, `linetab[0]` contiendra le nom du fichier et `linetab[1]` sa somme. Nous cherchons ensuite dans notre ligne de commande si la somme de notre fichier doit être vérifiée et, si tel est le cas, nous la vérifions. Si le fichier a subi une modification depuis son entrée dans la base, sa somme ne sera pas la même. Dans ce cas, nous affichons un message indiquant à l'utilisateur que le fichier a subi une modification.

```

void
check_sum_in_file(char *database_name, char **filename)
{
    int fd;
    char *line;
    char **linetab, **digest;
    int i;

    if ((digest = get_sum(filename))
        == NULL)
        return ;
}

```

```

if ((fd = open(database_name,
                O_RDONLY)) < 0) {
    perror("open");
    return ;
}

while ((line = getline(&fd)) {
    linetab = str_wordtab(line, '=');
    if (linetab[0] && linetab[1])
        for (i = 0; filename[i]; i++)
            if (!strcmp(filename[i],
                        linetab[0]))
                if (strcmp(digest[i],
                        linetab[1]))
                    printf("The file %s "
                            "has been modified\n", filename[i]);
}
}

```

Voilà, nous avons un petit contrôleur d'intégrité. Je dis petit car vous pouvez encore largement l'améliorer voire, pour ceux qui souhaitent aller plus loin, complètement le redévelopper selon vos méthodes ou vos besoins.

Les traitements par flux

Nous sommes maintenant de véritables spécialistes des appels système « `open` », « `write` » et « `read` », mais il faut savoir qu'il existe d'autres manières d'ouvrir, de lire et d'écrire dans un fichier. Les fonctions `fopen`, `fread` et `fwrite` réalisent le même travail, à ceci près qu'il s'agit de fonctions et non d'appels système et qu'au lieu de nous renvoyer un simple file descriptor, elles nous renvoient un « `stream` ».

```

FILE *fopen(const char *path,
            const char *mode);

```

Nous remarquons que les paramètres sont sensiblement les mêmes ; le chemin du fichier et les

droits avec lesquels doit être ouvert notre fichier. Ce n'est par contre pas des flags qui indiquent les droits mais simplement une chaîne :

« r » ouvre le fichier en lecture seule et « r+ » en lecture / écriture,

« w » crée le fichier pour l'écriture seulement s'il n'existe pas et l'écrit s'il existe,

« w+ » fait de même, sauf qu'il ouvre le fichier en lecture/écriture,

« a » correspond à :

O_APPEND|O_WRONLY|O_CREAT et enfin « a+ » revient à faire un classique « open » avec les flags O_APPEND|O_RDWR|O_CREAT.

Les fonctions fread et fwrite permettent quant à elles les mêmes traitements que leurs compères, les appels système « read » et « write ». C'est pourquoi je ne les détaillerai pas, mais je vous invite tout de même à vous reporter aux manuels de ces deux fonctions.

Les flux

Les « streams » dont je vous ai parlé plus haut correspondent à des flux.

Pour pouvoir utiliser un flux, il doit être associé à un périphérique ou un fichier. Vous pourrez lire et écrire dans le flux au même titre que dans un file descriptor.

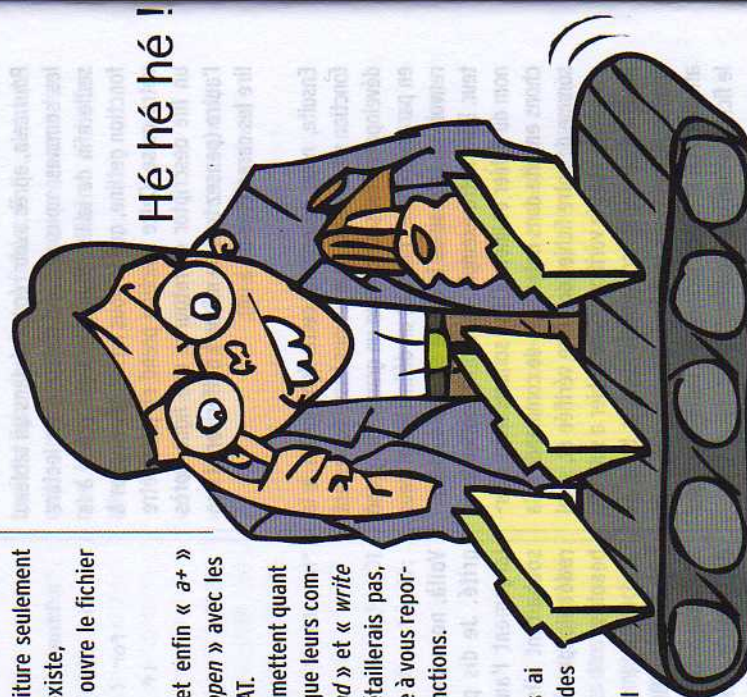
Il existe trois manières d'ouvrir un flux :

FILE *fopen(const char *path, const char *mode);

- fopen ouvre un flux donné avec les modes indiqués. Cette fonction est la plus utilisée pour ouvrir des fichiers.

FILE *fdopen(int fd, char *mode)

- fdopen assigne un flux à un file descriptor préalablement ouvert. Le mode d'ouverture du fd et de fdopen doit être le même.



Hé hé hé !

Vidage d'un flux

Parfois, lors de l'écriture d'un flux, il reste en attente dans une mémoire tampon. Pour éviter cela et « forcer » son écriture sur le disque, il peut s'avérer utile de « vider le flux ».

De même, il peut être pratique de pouvoir se débarrasser d'entrées utilisateurs pour en stocker de nouvelles (dans le cas d'un jeu par exemple). Il existe deux fonctions permettant de vider un flux.

L'une se débarrasse complètement des données alors que l'autre force simplement son écriture dans le flux.

int fflush(FILE *stream)

int fpurge(FILE *stream)

stream étant un flux existant préalablement ouvert.

Ces deux fonctions nous renvoient EOF (End Of File) lors d'un échec, sinon, elles renvoient 0.

- fflush est utilisée pour vider les tampons en écrivant les tampons du flux vers un périphérique ou un fichier.

- fpurge est utilisée pour vider les tampons des données non écrites ou non lues qui attendent dans un tampon. fpurge détruit complètement le tampon.

Fermeture de flux

Vous pouvez fermer un flux grâce à la fonction fclose. Au même titre que l'appel système « close » désassocie le file descriptor du fichier, fclose ferme le flux ouvert et désassocie du pointeur sur FILE:

int fclose(FILE *stream);

Où stream est le flux ouvert avec fopen.

Positionnement dans un flux

Au même titre qu'il est possible de se positionner à un endroit donné avec lseek, une fonction

l'autorise également pour les flux. Cette fonction c'est ... fseek.

Eh oui, après fopen, fclose, fread et fwrite, il aurait été étonnant que la fonction équivalente à lseek pour le traitement de flux s'appelle fseek. Il faut savoir qu'il est impossible de repositionner le flux d'un terminal, seuls les périphériques par blocs comme les disques durs l'autorisent.

Tout comme pour lseek, fseek affecte la position dans le fichier du flux à la valeur de l'offset plus la position indiquée par whence, qu'il s'agisse du début, de la position courante, ou de la fin du fichier.

D'autres fonctions utiles existent pour le positionnement dans un flux comme fgetpos, qui retourne la position courante du flux, mais je vous laisse consulter les manuels de ces fonctions si vous êtes intéressé.

Voilà, bien que cet article ne soit en rien exhaustif, j'espère au moins que vous aurez appris quelque chose et que vous pourrez maintenant réaliser des programmes manipulant des fichiers selon vos besoins, ou au moins que vous pourrez en parler avec vos amis :) Si vous avez des questions, des suggestions ou des remarques, n'hésitez surtout pas à m'envoyer un email.

Salut Thibaut

Références :

Les pages de manuel BSD

www.Linux-France.org

Et tant d'autres...

salut.thibaut@gmail.com

GESTION DES ENTRÉES CLAVIERS À LA LOUPE...

Il est intéressant de savoir gérer les événements claviers, dans le cas, par exemple, d'un shell ou encore d'un jeu tel qu'un snake. Si vous avez en vue de développer ce type d'applications, cet article vous apportera les méthodes pour pouvoir « intercepter » les événements claviers et les gérer ensuite à votre guise.

Gestion du tty

Vous vous êtes sûrement déjà aperçu que lorsqu'un programme lit sur l'entrée standard (stdin) avec des fonctions du type `scanf`, il va attendre que l'utilisateur frappe sur la touche entrée et va ensuite traiter la ligne. Cela s'appelle le « *mode canonique* ». Dans le cas, par exemple, d'un jeu de serpent, cela s'avère peu pratique. Imaginez : vous appuyez sur gauche puis entrée pour que votre serpent change sa direction, vous ne joueriez pas beaucoup je pense.

Notre terminal est géré par un driver, le pilote de terminal. Ce pilote a plusieurs rôles. Il se charge, notamment, d'afficher les caractères tapés au clavier, mais également d'interrompre un programme en cours d'exécution avec Control-C (la gestion des signaux), ou encore, de nous assurer une édition de ligne par exemple en permettant de revenir en arrière ou d'effacer un caractère dans un shell. Par défaut, le terminal fonctionne en mode canonique (il attend une frappe sur entrée pour traiter la ligne), mais il est possible de modifier son comportement, grâce notamment à l'appel système « *ioctl* ».

ioctl

L'appel système `ioctl` permet de paramétrer le comportement du driver. Son synopsis est le suivant :

```
int ioctl( int fd, unsigned long request, void *argp );
```

où `fd` est un file descriptor ouvert, `request` un

code auquel correspond généralement un `define`, et `argp` est dans la plupart des cas l'adresse d'une structure contenant les paramètres de notre terminal.

Il existe différents `define`s spécifiques à la gestion des terminaux :

`TIOCGETA` : récupère les attributs du tty (structure `termios`).

`TIOCSETA`, `TIOCSETAW`, `TIOCSETAF` : met en place les attributs du tty.

`TIOCGEOTD` : récupère la "line discipline" courante.

`TIOCSEOTD` : change la "line discipline".

Vous pouvez retrouver tous ces `define`s dans `/usr/include/sys/ttycom.h` sur BSD et dans `/usr/include/asm/linux/*` sous Linux. `ioctl` étant un appel système dont l'utilisation est loin d'être simplissime, des fonctions ont été implémentées pour les tâches les plus courantes réalisées avec `ioctl`, telles que les fonctions « *tcgetattr* » ou « *tcsetattr* » que nous verrons plus loin.

Termios

Dans la norme POSIX, tous les paramètres d'un terminal (et des liaisons séries) sont regroupés dans une structure appelée `termios` et définie dans le fichier « `termios.h` » qu'il nous faut donc inclure. Certains systèmes (poussiéreux ?) utilisent encore l'ancienne implémentation de `term`.

mios, à savoir `termio`, mais nous ne nous y intéresserons pas ici car nous considérons que la grande majorité d'entre vous utilisent Linux ou BSD ou n'importe quel autre Unix un peu moderne.

La structure `termios` est constituée comme suit :

```
struct termios {
    tcflag_t c_iflag;
    /* input mode flags */
    tcflag_t c_oflag;
    /* output mode flags */
    tcflag_t c_cflag;
    /* control mode flags */
    tcflag_t c_lflag;
    /* local mode flags */
    cc_t c_line;
    /* line discipline */
    cc_t c_cc[NCCS];
    /* control characters */
};
```

Un seul champ nous intéresse dans notre exemple, voici une présentation succincte des champs de cette structure. Vous êtes toutefois libres de lire le man de `termios` si vous souhaitez en savoir plus.

* `c_iflag` : Les modes d'entrée définissent un traitement à appliquer sur les caractères en provenance de la liaison série.

* `c_oflag` : Les modes de sortie à appliquer sur les caractères envoyés sur la liaison série.

* `c_cflag` : Les modes de contrôle permettant de définir le débit, la parité utilisée etc.

* `c_lflag` : Les modes locaux (celui qui nous intéresse). Il définit le mode (canonique ou non), la gestion de l'écho des caractères ou encore des signaux.

`ECHO` : Un écho des caractères reçus est effectué.

`ICANON` : Passage en mode canonique, tous les caractères seront immédiatement disponibles à la lecture.

`ISIG` : Certaines touches génèrent des signaux (Control-C ou Control-Z) (défini par défaut).

* `c_cc` : les caractères de contrôle.

C'est un tableau de caractères. Des constantes sont définies pour accéder à quelques éléments particuliers, seuls deux d'entre eux nous intéressent :

`VMIN` : En mode non-canonique, spécifie le nombre de caractères que doit contenir le tampon pour être accessible à la lecture. En général, on fixe cette valeur à 1.

`VTIME` : En mode non-canonique, spécifie, en dixièmes de seconde, le temps au bout duquel un caractère devient accessible, même si le tampon ne contient pas `c_cc[VMIN]` caractères. Une valeur de 0 représente un temps infini.

Un peu de pratique ...

Après toute cette théorie, un peu de pratique s'impose pour éclaircir tout cela. Nous allons donc commencer par programmer une fonction qui va se charger de passer notre terminal en mode non-canonique (les caractères ne sont pas traités par lignes), avec `ioctl`, ça donnerait :

```
int main(void)
{
    /* non_canonic_mode(void)
    /* Notre structure contenant les
    /* informations du terminal */
    struct termios term;

    if (ioctl(0, TIOCGTA, &term)) {
        /* Recuperation des informations de
        /* la structure termios */
        perror("ioctl");
        return (-1);
    }
    term.c_lflag &= ~(ICANON|ECHO);
    /* Initialisation du terminal en
    /* mode non-canonique, avec echo des
```

caracteres a l'ecran. Nous pourrions intercepter les signaux en ajoutant le flag SIG * /

```
term_cc[VMIN] = 1;
/* 1 seul caractere dans le tampon */
term_cc[VTIME] = 0;
/* Caracteres accessibles immediatement */

if (ioctl(0, TIOCGSETA, &term)) {
    /* Mise a jour de notre structure */
    perror("ioctl");
    return (-1);
}

return (0);
}
```

Comme je l'ai dit plus haut (vous avez suivi n'est-ce pas ?), des fonctions permettant de récupérer les informations du terminal et de les mettre à jour sont disponibles sur tous les Unix modernes. Il s'agit de `tgetatr()` et `tsetatr()`. Nous pourrions ainsi remplacer respectivement les lignes `ioctl(0, TIOCGSETA, &term)` par `tgetatr(0, &term)` et `ioctl(0, TIOCSSETA, &term)` par `tsetatr(0, TCSANOW, &term)`.

Gestion des touches

Maintenant que nous savons initialiser notre terminal, il devient possible de lire les événements clavier et de les gérer. Pour cela, nous allons faire une fonction qui va lire sur l'entrée standard et retourner la valeur de la touche tapée. Il nous faut donc savoir que les touches ne renvoient pas toutes le même nombre d'octets. En effet, les touches telles que les lettres renvoient un « scan code » alors que lorsque vous enfoncez une touche telle que les flèches directionnelles, elle n'en-

voie pas un scan code mais deux ou trois. C'est pour cela que notre fonction devra faire des `read` successifs jusqu'à la fin de la lecture de ce code.

```
unsigned int readchar(void)
{
    unsigned int n;
    unsigned char c;

    read(0, &c, 1);
    /* Lecture du premier octet */
    n = c;
    /* On affecte a n la valeur de c */

    if (n == 0x1b) {
        /* s'agit-il d'une touche directionnelle ? */

        read(0, &c, 1);
        /* Lecture du 2eme octet */
        n = (n << 8) | c;
        /* Traitement du caractere */
        read(0, &c, 1);
        /* Lecture du troisieme octet */
        n = (n << 8) | c;

        if ((c & 0xf0) == 0x30) {
            /* On continue jusqu'a ce que tous les octets soient lus */
            read(0, &c, 1);
            n = (n << 8) | c;
            if ((c & 0xf0) == 0x30) {
                read(0, &c, 1);
                n = (n << 8) | c;
            }
        }
        return (n);
    }
}
```

Il nous reste maintenant à réaliser une fonction qui va se charger d'afficher à l'écran le code des touches, ce sera effectué dans la fonction `main()`.

```
int main(int argc, char *argv[])
{
    unsigned int val;
    non_canonic_mode();
    while((val = readchar())) {
        printf("%x\n", val);
    }
}
```

Nous avons maintenant un programme qui lit les touches et affiche leur code (super non ?) ! Nous pourrions encore l'améliorer, par exemple, en affichant le nom des touches plutôt que leur valeur. Nous allons donc utiliser un petit tableau de pointeur sur fonctions.

```
typedef struct vals {
    /* Définitions de la structure */
    unsigned int key_code;
    void (*f)();
} val_t;

#define UP_CODE 0x001b5b41
/* Les valeurs des touches lues avec readchar */
#define DOWN_CODE 0x001b5b42
#define RIGHT_CODE 0x001b5b43
#define LEFT_CODE 0x001b5b44

void fonction_down(void) /* Nos fonctions associées aux touches */
{
    printf("down\n");
}
```

Ensuite, nous modifions notre main :

```
int main(void)
{
    unsigned int i, valkey;
    val_t valtab[] = {
        {LEFT_CODE, fonction_left},
        {UP_CODE, fonction_up},
        {RIGHT_CODE, fonction_right},
        {DOWN_CODE, fonction_down},
        {0, 0}
    };
    non_canonic();
    while ((valkey = readchar())) {
        /* Recup. du code de la touche */
        for (i = 0; valtab[i].key_code; i++) {
            /* Parcours du tableau */
            if (valtab[i].key_code == valkey) {
                /* Si les valeurs matchent, on appelle la fonction associée */
                valtab[i].f();
            }
        }
    }
}
```

Voilà, nous avons un programme qui permet de lire les touches du clavier et d'appeler la fonction que l'on désire lui voir associée.

Dans le cas du développement d'un snake en termcaps ou encore d'un shell, cela peut s'avérer utile :-)

Je vous laisse donc le loisir de modifier à votre gré ce petit programme et développer vous mêmes les fonctions que vous voulez attribuer aux touches. En espérant que vous avez compris et que vous saurez vous en resservir à bon escient.

Salut Thibaut
salut.thibaut@gmail.com

DÉCOUVREZ UNE EXPLOITATION ORIGINALE DE BUFFERS OVERFLOWS

La méthode que je présente a ceci d'originale qu'elle répond aux problèmes rencontrés lors de l'exploitation d'un buffer overflow sur un contexte particulier de la pile. Elle a pour origine un arrachage de cheveux mineur lors d'une exploitation...

Le contexte et le problème

Les difficultés techniques telles que je vous les présenterai sont courantes, nous verrons que le contexte exemple que nous étudierons s'inscrit dans une problématique générale facilement abordable.

La méthode que nous décortiquerons ensuite permet de pallier aux difficultés en toute simplicité et avec une grande efficacité. D'où le sommaire. Suivez.

La lecture de ce tutorial présume que vous savez exploiter un stack overflow. Disons autrement que si vous savez exploiter un stack overflow, vous comprendrez cet article.

Zut... vous êtes partis ?

Ha non ? Vous êtes toujours là, très bien. Etudions un petit bout de code "scolaire", et situons-y la vulnérabilité.

```

--- Exemple ---
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void copy(char *str)
{
    char buf[64];
    int i;
    memset(buf, 0, 64);

```

(l'environnement d'exécution) nous empêche de brutalement forcer, pour quelque raison que ce soit, l'adresse de retour sur un éventuel shellcode passé en argument.

L'idée de base est donc de passer par une méthode efficace où la prédiction joue un rôle clef, par exemple un return-into-libc.

Voilà posée la requête : exploiter un return-into-libc sur ce code. Pourquoi ? Pour profiter du fait que cette méthode est beaucoup plus précise que le bruteforce - surtout quand le buffer ne fait que 64 octets.

Si vous ne connaissez pas le principe d'un return-into-libc, lisez l'annexe consacrée à l'exploitation de cette méthode. Revenez ici plus tard. Non pas ici, deux paragraphes au dessus. Je postule désormais que vous comprenez qu'il est techniquement nécessaire de préparer un certain nombre de données sur la pile afin de voir aboutir un return-into-libc. Alors soyez attentifs à ce qui suit...

Zoomons le code de la fonction copy(). L'adresse de la chaîne à copier est le seul argument à être passé à la fonction. Autrement dit, c'est le premier argument, il se trouve juste après EIP.

Si l'élément provoquant le buffer overflow - ici une chaîne trop longue - est l'un des premiers passés en argument, alors il devient impossible de l'écraser pour placer d'autres données sur la pile. Malheureusement ces "autres données" sont justement celles qui nous servent à réaliser un return-into-libc.

Comprenez-vous bien pourquoi un return-into-libc nous est interdit ? Le fait d'écraser le pointeur qui est utilisé pour la copie va détourner la copie sur

une autre donnée que la chaîne de caractères originale, et en parallèle, la copie continue d'écraser le pointeur à partir de données aléatoires.

Pour chaque octet copié sur le pointeur, les données concernées par celui-ci sautent... Danse folle qui se termine sur un octet nul ou une erreur de segmentation.

Solution : le ret-into-ret

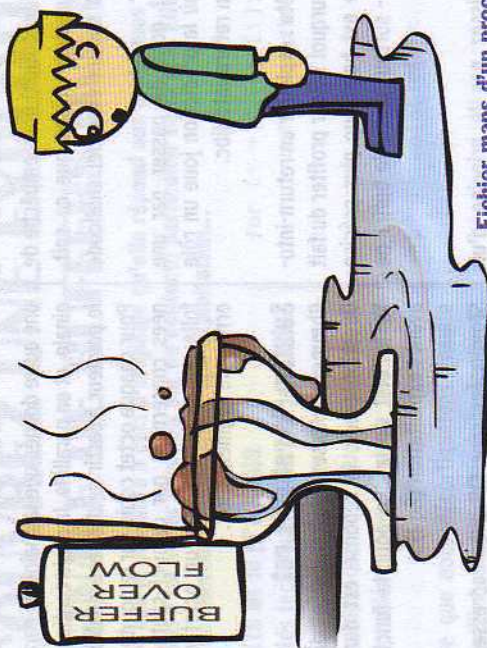
Comment passer outre cette difficulté ? Question piège, la réponse est dans le titre. Voilà l'état de la pile pour la fonction copy() avant l'exploitation (fig 1) :

Si nous écrasons EIP avec l'adresse d'une instruction ret, un deuxième retour de fonction sera exécuté.

haut	Ce deuxième retour va prendre l'adresse pointée par le registre ESP, à savoir *str, et la placer dans EIP.
...	L'exécution continue sur notre chaîne de caractères, qui pourrait tout aussi bien contenir un shellcode.
*str	Comme l'exécution revient directement sur le début de notre chaîne, nous pouvons profiter pleinement
EIP	des 64 octets du buffer pour placer un shellcode, sans avoir besoin d'instructions de type NOP, et en présupposant, bien sûr, que la pile soit exécutable.
buf[64]	La prédiction est ici assurée automatiquement par la configuration de la pile. Un seul retour est ici nécessaire, car l'argument qui pointe notre shellcode est le premier. On peut envisager plusieurs retours pour faire coïncider EIP et *str si ce pointeur n'est pas le premier, mais le deuxième ou troisième argument passé à la fonction vulnérable.
...	
bas	

Si l'élément provoquant le buffer overflow - ici une chaîne trop longue - est l'un des premiers passés en argument, alors il devient impossible de l'écraser pour placer d'autres données sur la pile. Malheureusement ces "autres données" sont justement celles qui nous servent à réaliser un return-into-libc.

Comprenez-vous bien pourquoi un return-into-libc nous est interdit ? Le fait d'écraser le pointeur qui est utilisé pour la copie va détourner la copie sur



Fichier maps d'un processus dans / proc

08048000-0804c000	r-xp	00000000	03:01	589888	/bin/cat
0804c000-0804d000	rw-p	00003000	03:01	589888	/bin/cat
0804d000-0806e000	rw-p	0804d000	00:00	0	
40000000-40016000	r-xp	00000000	03:01	671776	/lib/ld-2.3.2.so
40016000-40017000	rw-p	00015000	03:01	671776	/lib/ld-2.3.2.so
40017000-40018000	rw-p	40017000	00:00	0	
40022000-4014b000	r-xp	00000000	03:01	2048049	/lib/tls/libc-2.3.2.so
4014b000-40153000	rw-p	00129000	03:01	2048049	/lib/tls/libc-2.3.2.so
40153000-40157000	rw-p	40153000	00:00	0	
bffff000-c0000000	rw-p	bffff000	00:00	0	
ffffe000-fffff000	---p	00000000	00:00	0	

Toute la difficulté réside dans la recherche de l'adresse d'une instruction ret, ce qui n'est d'ailleurs pas réellement une difficulté en soi : la libc en contient assez pour toute une armée de shellcoders. On applique les méthodes habituelles de prédiction si on n'a nourri pas la main sur le profs du système local. A moins que...

Ret-into-vsyscalls sur Linux 2.6

Par chance, les développeurs du kernel ont eu la bonté de prendre en compte nos tracas, et ont intégré sur le noyau 2.6 un support pour les appels système virtuels nativement activés.

Cet article est, pour l'ignorant, l'occasion d'en apprendre un peu plus sur les vsyscalls.

Les appels système virtuels se présentent sous la forme de code exécutable mappé en espace noyau, mais dont le code est accessible depuis un processus user-land. Ils ont pour but d'accélérer le temps d'exécution de fonctions "basiques", normalement assurées par le noyau, mais dont le code pourrait tout aussi bien être exécuté sans passer par une interruption coûteuse en cycles processeurs.

C'est par exemple le cas de la fonction (man 2) sigreturn() qui a son équivalent en vsyscall.

Vous l'aurez compris, si le code est exécutable en

user-land, alors la région mémoire du code est mappée pour le processus. Confirmons (fig 2) : La région mémoire mappée à 0xffffe000 contient nos vsyscalls... Et peut-être un ret.

On attache un processus avec GDB, on désassemble la région mémoire concernée, et qu'on obtient-on en louchant un peu ?

```
(gdb) x/1 0xfffffe413
0xfffffe413 <_kernel_vsyscall+19>: ret
```

Il s'avère que sur de nombreux noyaux 2.6 vous pouvez être sûrs de trouver un return à cette adresse. A vérifier chez vous.

Voilà une utilisation bien pratique des vsyscalls.

Conclusion

Partant d'un problème simple, nous avons trouvé une solution élégante pour retourner sans aucune difficulté sur du code exécutable placé sur la pile (ou dans le tas), en écrasant à partir de EIP, voire au delà, zero, un ou plusieurs arguments avec une adresse redondante pointant sur une instruction ret.

Cet article est extrait du magazine nOname.
Merci à eux.

Le saviez vous ?

Les vsyscalls ont été implémentés à partir de la version 2.6 du noyau Linux. Ils n'ont pas d'équivalent sur les autres OS. Ils vont permettre aux applications d'accélérer grandement leur temps d'exécution dans les appels successifs de fonctions système non critiques (par exemple gettimeofday()). On reprochait en effet à certains appels systèmes de beaucoup trop ralentir le temps d'exécution de programmes sur des processeurs supposés puissants. Un petit papier d'Arcangeli reprend en résumé des informations intéressantes à ce sujet : <http://www.ukuug.org/events/linux2001/papers/html/AArcangeli-vsyscalls.html>

LES BASES D'OPENGL

Introduction

OpenGL a été inventé par SGI (Silicon Graphics Integrated) pour faire du rendu d'image en trois dimensions très rapidement. L'application appelle des fonctions qui vont envoyer des commandes à la carte graphique, qui à son tour va modifier la mémoire vidéo et donc permettre l'affichage.

OpenGL est en réalité une spécification, c'est à dire que c'est un document décrivant quelles fonctions sont disponibles et comment elles doivent se comporter. Cette spécification est libre, ainsi tout le monde peut programmer une bibliothèque qui fournit les fonctionnalités de cette spécification. Suivant le système sur lequel vous travaillez, vous aurez accès à différentes implémentations, mais elles font toutes la même chose.

Utiliser OpenGL est assez simple lorsque l'on a compris comment le dialogue se fait avec la carte graphique. Mais un problème n'est pas résolu par OpenGL : l'initialisation. Ceux qui ont pensé OpenGL

se sont dit qu'initialiser une fenêtre graphique était dépendant du système et n'avait rien à voir avec du rendu 3D haute performance. Il faut donc utiliser d'autre fonctionnalités du système pour démarrer OpenGL (wgl sur Windows, glx sur UNIX).

Cependant, certaines personnes ont créé des bibliothèques qui permettent ceci et qui fonctionnent sur tous les systèmes, par exemple Glut ou SDL. Nous allons utiliser SDL dans notre exemple.

Pour introduire OpenGL et démontrer sa simplicité, nous allons prendre l'exemple du "cube qui tourne".

```

-- main.c --
#include <SDL/SDL.h>
#include <GL/GL.h>

SDL_Surface *screen = NULL;

int main(int argc, char *argv[])
{
    demarre_opengl();
    tourne_cube();
    stop_opengl();
}

--fin main.c --

```

L'initialisation

L'initialisation consiste en l'implémentation de la fonction `demarre_opengl()`.

Il faut initialiser SDL puis demander une fenêtre (ou un plein écran) capable de contenir un affichage OpenGL. Il faut enfin initialiser et appeler des fonctions OpenGL pour mettre la carte graphique dans le mode que l'on veut pour le rendu de notre cube.

```

-- main.c --

void demarre_opengl()
{
    creer_fenetre_opengl();
    init_opengl();
}

```

```

void creer_fenetre_opengl()
{
    SDL_Init(SDL_INIT_VIDEO);

    SDL_GL_SetAttribute(
        SDL_GL_RED_SIZE, 5);

    SDL_GL_SetAttribute(
        SDL_GL_GREEN_SIZE, 5);

    SDL_GL_SetAttribute(
        SDL_GL_BLUE_SIZE, 5);

    SDL_GL_SetAttribute(
        SDL_GL_DEPTH_SIZE, 16);

    SDL_GL_SetAttribute(
        SDL_GL_DOUBLEBUFFER, 1);

    screen = SDL_SetVideoMode(
        800, 600, 16,
        SDL_OPENGL
        | SDL_DOUBLEBUF
        | SDL_HWSURFACE);

    void init_opengl()
    {
        /* utilise tout l'ecran */
        glViewport(0, 0, 800, 600);

        /* les couleurs sont dégradées */
        glShadeModel(GL_SMOOTH);

        /* couleur (noir) lorsque l'on
        efface l'ecran */
        glClearColor(0.0, 0.0, 0.0,
            1.0);

        /* profondeur des pixels lorsque
        l'on efface l'ecran */
        glClearDepth(1.0);

        /* comment choisir si un point
        est devant un autre ou pas */
        glDepthFunc(GL_LEQUAL);

        /* n'affiche pas les points
        caches */
        glEnable(GL_DEPTH_TEST);

        /* initialisation de la base
        (repere en 3 dimension) */
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        /* toutes les faces d'un polygone
        doivent etre dessinees */
        glPolygonMode(
            GL_FRONT_AND_BACK, GL_FILL);
    }

    void stop_opengl()
    {
        SDL_Quit();
    }

    -- fin main.c --

```

L'initialisation d'OpenGL peut paraître un peu obscur de prime abord, mais c'est en fait assez intuitif, les commentaires insérés dans le code suffisent. Si vous ne comprenez pas tous les détails, ce n'est pas grave car cette procédure d'initialisation n'a pas besoin d'être changée. Reportez-vous à un livre spécialisé si vous voulez plus de détails.

Le cube qui tourne

Enfin nous y arrivons ! Commençons pas regarder le code puis il sera décortiqué pour comprendre les détails.

```
-- main.c --
```

```
void tourne_cube()
```

```
{
```

```
    int i = 1000;
```

```
    while (i--)
```

```
    {
```

```
        SDL_Delay(30);
```

```
        glClearColor(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

```
        glBegin(GL_QUADS);
```

```
        /* face du dessous */
```

```
        glColor3f(1.0, 0.0, 0.0);
```

```
        glVertex3d(-0.3, -0.3, -0.3);
```

```
        glVertex3d(-0.3, -0.3, 0.3);
```

```
        glVertex3d(0.3, -0.3, 0.3);
```

```
        glVertex3d(0.3, -0.3, -0.3);
```

```
        /* face de devant */
```

```
        glColor3f(0.0, 1.0, 0.0);
```

```
        glVertex3d(-0.3, -0.3, -0.3);
```

```
        glVertex3d(-0.3, 0.3, -0.3);
```

```
        glVertex3d(0.3, 0.3, -0.3);
```

```
        glVertex3d(0.3, -0.3, -0.3);
```

```
        /* face du dessus */
```

```
        glColor3f(1.0, 0.0, 0.0);
```

```
        glVertex3d(-0.3, 0.3, -0.3);
```

```
        glVertex3d(-0.3, 0.3, 0.3);
```

```
        glVertex3d(0.3, 0.3, 0.3);
```

```
        glVertex3d(0.3, -0.3, 0.3);
```

```
        /* face de derriere */
```

```
        glColor3f(0.0, 1.0, 0.0);
```

```
        glVertex3d(-0.3, -0.3, 0.3);
```

```
        glVertex3d(-0.3, 0.3, 0.3);
```

```
        glVertex3d(0.3, 0.3, 0.3);
```

```
        glVertex3d(0.3, -0.3, 0.3);
```

```
        /* face de droite */
```

```
        glColor3f(0.0, 0.0, 1.0);
```

```
        glVertex3d(-0.3, -0.3, -0.3);
```

```
        glVertex3d(0.3, -0.3, -0.3);
```

```
        glVertex3d(0.3, 0.3, -0.3);
```

```
        glVertex3d(0.3, 0.3, 0.3);
```

```
        glVertex3d(-0.3, 0.3, 0.3);
```

```
        /* face de gauche */
```

```
        glColor3f(0.0, 0.0, 1.0);
```

```
        glVertex3d(-0.3, -0.3, -0.3);
```

```
        glVertex3d(-0.3, 0.3, -0.3);
```

```
        glVertex3d(-0.3, 0.3, 0.3);
```

```
        glVertex3d(-0.3, -0.3, 0.3);
```

```
        glEnd();
```

```
        glRotated(5, 1.0, 1.0, 1.0);
```

```
        SDL_GL_SwapBuffers();
```

```
    }
```

```
    -- fin main.c --
```

Explications

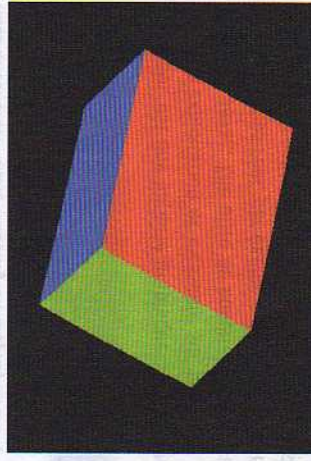
La variable `i` est là pour dire combien de fois on veut dessiner le cube, de telle sorte que notre programme s'arrête quand il a fini.

La fonction `SDL_Delay` permet d'attendre un certain nombre de millisecondes.

`glClearColor(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)` dit à OpenGL d'effacer la mémoire qui contient l'information de couleur (celles que l'on voit) et la mémoire qui contient l'information de profondeur (pour déterminer quel Vertex est devant un autre et donc, ne pas afficher ceux qui sont derrière).

`glBegin(GL_QUADS)` dit à OpenGL que l'on va dessiner des carrés, d'autres valeurs sont possibles telles que `GL_TRIANGLES`, `GL_LINES`...

`glEnd()` indique à OpenGL que nous avons fini de dessiner, que les commandes peuvent alors être envoyées à la carte graphique.



`glVertex3d(...)` dit à OpenGL que l'on place aux coordonnées indiquées un vertex, c'est à dire un point en 3 dimensions, dans ce contexte, un des coins du carré. Remarquez que chaque vertex est décrit trois fois car

chaque coin est partagé par trois faces. Il y a une manière d'optimiser ceci en passant `GL_TRIANGLE_STRIP` à `glBegin()`.

`glRotated(...)` demande à OpenGL de faire une rotation du repère, ce qui a pour conséquence que la prochaine fois que l'on dessinera la

RETURNS-INTO-LIBC : C'EST DANS LA POCHE !

Cet article constitue un rappel sur les **buffer overflows exploitables par return-into-libc**. Sa lecture s'adresse aux personnes sachant au moins exploiter un **stack overflow standard**.

La méthode du **return-into-libc** consiste, non pas à exécuter un shellcode, mais à détourner le programme en lui faisant exécuter du code qui, comme vous l'avez compris, est bien souvent dans une librairie telle que la lib.

Et comme les librairies sont chargées à des adresses prédictibles sur les noyaux standards, on les retrouve très facilement d'un programme à l'autre.

La lib, par exemple, contient toutes les fonctions utiles à l'exécution, par exemple, d'un shell. Un simple appel à `(man 3) system()`, avec l'adresse d'une chaîne `"/bin/sh"` ne suffirait-il pas à satisfaire certains de nos besoins ? (Si oui, c'est un aveu !)

Il faut que vous visualisiez le processus suivant : vous venez d'écraser EIP avec l'adresse de la fonction `system()` située dans l'espace mémoire du processus vulnérable. Le programme saute sur `system()`. La fonction système va faire son prologue, sans erreur. Par contre, `system()` va vouloir aller chercher le paramètre que vous lui avez passé, et il s'attend pour cela à ce qu'il y ait un argument sur la pile.

L'argument sur la pile, c'est l'adresse d'une chaîne de caractères `"/bin/sh"`. Seulement, pas de bol, elle n'est pas là. Il faut écrire cet argument sur la pile si vous voulez que `system()` le récupère.

Et où `system()` va-t-il récupérer ce fameux pointeur ? Juste après EIP bien sûr. La convention d'appel de fonctions veut que les arguments soient empilés dans l'ordre inverse de leur déclaration en C, juste

alors rajoutez manuellement la ligne `Load "glx"` dans le fichier de configuration de X et redémarrez-le.

Si vous utilisez Windows avec DevC++, il faut lui demander d'ajouter ceci à la ligne de commande dans les options projet. Si vous utilisez VisualC++ ou équivalent, il suffit d'ajouter `libopengl.dll`, `libSDL.dll` et `libSDLmain.lib` dans les bibliothèques à utiliser dans les options projet.

Pour aller plus loin dans OpenGL, toute la documentation est disponible sur Internet. Vous êtes maintenant prêt à vous lancer dans la programmation 3D en OpenGL.

Philippe Amarengo

scène, on la verra d'un autre angle. On aura alors l'impression que le cube tourne.

`SDL_GL_SwapBuffers()` demande à SDL d'échanger la mémoire dans laquelle on dessine avec la mémoire que l'on affiche. Ceci permet de ne pas voir les étapes intermédiaires lors de l'étape de dessin.

Compilation

Pour compiler ce programme, ajoutez

```
-lGL `sdl-config --cflags` -lSDL -lSDLmain
```

sur la ligne de commande de gcc si vous utilisez gcc sous UNIX (Linux, *BSD, OSX...).

Note : si vous avez comme message d'erreur `Xlib: extension "GLX" missing on display ":0.0"`,

Le saviez-vous ?

OpenGL est une librairie graphique 3D très évoluée, parfaitement portable sur tous les systèmes d'exploitation et utilisable sur plusieurs langages de programmation tels que le C/C++, Delphi, Java... Elle offre de nombreuses ressources aux programmeurs désirant faire un rendu 3D en temps réel, par exemple pour un moteur 3D pour un jeu vidéo.

En effet, OpenGL est portable sur tous les systèmes d'exploitation car ce n'est pas elle qui se charge de l'affiche. Cette librairie fait "simplement" le lien entre le programme et le système d'exploitation, celui-ci renvoie par la suite les instructions à la carte graphique par l'intermédiaire d'un driver (`opengl32.dll` pour Windows). On a ainsi accès à de nombreuses fonctions matérielles de la carte graphique, qui sont beaucoup plus performantes qu'un affichage purement logiciel. En pratique, le programmeur sous Windows devra tout d'abord initialiser un environnement graphique, donc une fenêtre. OpenGL doit ensuite s'adresser à cette fenêtre pour lui envoyer ses ordres de dessin. Côté utilisation, OpenGL offre toute une panoplie de fonctions pour gérer un affichage en 3D temps réel. Ainsi, même les tâches les plus complexes comme la gestion des textures ou des lumières, par exemple, sont réduites à l'utilisation de quelques fonctions.

Il existe une autre librairie semblable à OpenGL, c'est Direct 3D de Microsoft. Son mode de fonctionnement est quasiment le même. Pour débuter, il est tout de même préférable d'utiliser OpenGL car bien plus simple d'utilisation, de plus OpenGL est open source et donc gratuite contrairement à sa rivale :)

AVANT :	APRES :
Haut	Haut
...	...
QQCH2	QQCH2
QQCH1	"/bin/sh"
ARG1	fake EIP
EIP	system()
...	...
Bas	Bas

- URG: indique si le segment est en mode urgent.
- ACK: indique que le segment sert à l'acceptation d'une connexion TCP.
- PSH: flag « push ».
- RST: réinitialise la connexion.
- SYN: indique qu'il faut synchroniser la connexion.
- FIN: indique qu'il n'y a plus de données à échanger, donc met fin à la connexion.
- « Window » (16 bits): le nombre d'octets de données que la machine émettrice est capable d'accepter.
- « Checksum » (16 bits): la somme de divers champs de l'entête TCP.
- « Urgent pointer » (16 bits): définit où se terminent les données urgentes.
- « Options » (variable): champ qui permet de spécifier des options.
- « Padding » (variable): sert à rajouter des bits à 0 pour faire commencer les données au début d'un mot de 32 bits.
- « Data » (variable): les données d'un segment TCP.

Les sockets

Qu'est-ce qu'une socket ? Au sens propre, c'est une « prise ». Mais c'est plus vraisemblablement un canal de communication qui permet de relier deux ordinateurs à travers un réseau.

Cette connexion, sous TCP, permet un transfert fiable des données. De plus, le système possède tout un système de gestion d'erreurs qui permet au programmeur de ne pas se soucier des détails de la connexion.

Sous Unix, ce tuyau est représenté par un fichier. Mais ils sont légèrement différents des fichiers normaux.

Au lieu d'avoir un « - » comme type de fichier, ils ont un « s ». La plupart du temps, ils se trouvent dans le répertoire /tmp. Cherchez un peu pour les trouver, ils se cachent souvent dans des répertoires. Par exemple :

Ne vous affolez pas... Tous ces champs ont une signification. Il vous faut les connaître (ou au moins les avoir vu), ainsi que leur taille. Ceux qui les connaissent déjà peuvent attaquer sans remords la seconde partie.

- « Source port » (16 bits): le port source, celui de la machine locale.
- « Destination port » (16 bits): le port de la machine qui va recevoir le segment.
- « Sequence number » (32 bits): un numéro généré qui permet aux deux machines de communiquer avec un séquençage correct des segments. Cela permet de gérer des erreurs (un segment qui manque, des segments en doublons...).

Lorsque le flag SYN est activé, c'est le premier champ du segment.

- « Acknowledgment number » (32 bits): si le flag ACK est activé, ce champ contient le prochain numéro du segment attendu.

- « Data offset » (4 bits): le nombre de mots de 32 bits contenus dans l'entête du segment TCP. L'entête ne comprend pas le champ data.

- « Reserved » (6 bits): ces bits sont réservés à une utilisation ultérieure. Ils devraient être à zéro.

- Les « control bits » (6 bits) (de gauche à droite, chacun faisant 1 bit) :

```
Raf@lucifer /tmp > ls -la
total 8
drwxrwxrwt 3 root wheel 512 ./
drwxr-xr-x 20 root wheel 512 ../
drwxr-xr-x 2 root wheel 512 .X11-unix/
Raf@lucifer /tmp > cd .X11-unix/
Raf@lucifer /tmp/.X11-unix > ls -l
total 0
srwxrwxrwx 1 root wheel 0 X0=
Raf@lucifer /tmp/.X11-unix >
```

Comme vous le voyez, cette socket possède « s » comme premier attribut. Ils se retrouvent parfois dans des répertoires cachés ou non. Certaines se retrouvent directement dans le répertoire /tmp.

Concrètement, pour envoyer des données à travers une socket, il suffit juste d'écrire dans le fichier. Mais comme ce fichier possède quelques particularités, il faut utiliser des appels systèmes pour le manipuler...

Le mode « promiscuous »

Le mode promiscuous est un état particulier de l'interface réseau. Normalement, une interface réseau ne récupère que les segments qui lui sont destinés. Elle défasse tout le reste.

Si nous mettons l'interface en mode « promiscuous », elle récupérera tous les segments, même ceux qui ne lui sont pas destinés. C'est pratique dans des environnements de réseaux locaux.

L'appel système ioctl

Cet appel système permet de faire énormément de choses sur les systèmes Unix et Linux, mais nous allons l'utiliser uniquement pour configurer une interface réseau. La macro SIOCGIF-FLAGS sert à récupérer la configuration d'une interface réseau.

La macro SIOCSIFFLAGS permet de configurer une interface réseau suivant les paramètres saisis. C'est avec cet appel système que l'on va pouvoir passer l'interface réseau en promiscuous.

La réception des segments TCP

Tout ce qui est expliqué ci-dessus permet de comprendre le listing ci-dessous. Plutôt qu'un long discours, voici le listing en question :

```
sniffer.c
/* En-têtes */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/route.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

/* On réécrit la structure d'une entête IP (Vous pouvez la réécrire ou utiliser <linux/ip.h> si vous préférez aux codes du pingouin)
[type var:n] déclare une variable de n bits */
struct ip
{
    /* little-endian */
    unsigned int ip_length;
    unsigned int ip_version;
    unsigned char ip_tos;
    unsigned short ip_total_length;
    unsigned short ip_id;
    unsigned short ip_flags;
    unsigned char ip_ttl;
};
```

```

unsigned char ip_protocol;
unsigned short ip_cksum;
unsigned int ip_source;
unsigned int ip_dest;
};

/* Structure d'un entete TCP (ou cf
aussi <linux/tcp.h>) */
struct tcp {
    unsigned short tcp_source_port;
    unsigned short tcp_dest_port;
    unsigned int tcp_seqnum;
    unsigned int tcp_acknum;

    /* little-endian */
    unsigned int tcp_res1:4;
    tcp_hlen:4;
    tcp_fin:1;
    tcp_syn:1;
    tcp_rst:1;
    tcp_psh:1;
    tcp_ack:1;
    tcp_urg:1;
    tcp_res2:2;

    unsigned short tcp_winsize;
    unsigned short tcp_cksum;
    unsigned short tcp_urgent;
};

/* On annonce le prototype */
int mode_promiscuous(char *inter-
face, int sock);

int main(int ac, char **av)
{
    int sock; /* descripteur de
    Les 4 octets du fichier */
    int octets_recus;

```

```

    sur une socket et met le resultat
    dans le deuxieme argument. */
    octets_recus =
    recvfrom(sock, buf,
    sizeof(buf), 0,
    (struct sockaddr *)&segment,
    &segment_taille);
    printf("\nOctets recus : %5d\n",
    octets_recus);

    /* La fonction inet_ntoa()
    sert a convertir une
    adresse internet en chaine
    de caracteres ASCII. */
    printf("Adresse source : %s\n",
    inet_ntoa(segment.sin_addr));

    /* Le paquet est dans: buf, on
    fait pointer dessus : on
    redefinit le pointeur buf en
    une structure IP. Cela per-
    met de lire l'entete IP. */
    ip = (struct ip *) buf;

    /* Pour lire l'entete TCP, il
    faut redefinir a nouveau
    le pointeur. Nous connais-
    sons la taille de l'entete
    IP a deduire pour obtenir
    le debut du segment TCP.
    Nous n'avons pas affiche
    tous les champs, car a
    part les ports, ils ne
    sont pas utiles a obser-
    ver. Par contre, le champ
    data est certainement le
    plus interessant. Il faut
    decaler de la taille de
    l'entete IP et de la
    taille de l'entete TCP

```

```

    pour obtenir le champ
    data. Avec printf(), vous
    ne recupererez pas toutes
    les donnees s'il y a un
    caractere '\0'. */
    if(ip->ip_protocol == 6) {
        printf("Longueur de "
        "l'entete IP : %d\n",
        ip->ip_length);
        tcp = (struct tcp *)
        (buf + (4*ip->ip_length));

        /* La fonction ntohs() sert
        a convertir les donnees
        du segment dans le for-
        mat de la machine sur
        laquelle nous sommes. */
        printf("Port source : %d\n",
        ntohs(
            tcp->tcp_source_port));
        printf("Port destination : %d\n",
        ntohs(tcp->tcp_dest_port));
        data = (char *)
        (buf + (4*ip->ip_length)
        + (4*tcp->tcp_hlen));
        printf("data = %s\n",
        data);
    }

    int mode_promiscuous(
    char *interface, int sock)
    {
        struct ifreq ifr;
        struct ifreq ifr;
        strncpy(ifr.ifr_name, interface,
        strlen(interface) + 1);
        if(ioctl(sock, SIOCGIFFLAGS,

```

DEVENEZ FORGERON DE PAQUETS TCP SOUS GNU/LINUX

Il est grisant de pouvoir contrôler la construction des paquets du réseau en passant par les modes raw socket. Alors grisez, grisez, mais que votre chapeau reste blanc !

Introduction

Dans ce tutoriel, vous apprendrez comment faire un forgeur de paquet TCP rudimentaire. Mais avant d'entrer dans le vif du sujet vous aurez besoin de certaines bases en C et sur les datagrammes TCP et IP (cf. Article sniffing TCP). Ce tutoriel se fonde sur les « raw sockets ». Il s'agit d'un concept qui permet de créer des datagrammes à la volée en laissant le soin au développeur de spécifier les informations des entêtes de protocole.

Ne tardons pas dans le blabla, entrons directement dans le vif du sujet.

Pour pouvoir programmer notre forgeur de paquet il faut d'abord savoir comment notre programme devra s'exécuter.

Ce programme devra prendre en paramètre l'adresse IP destination et source, le type de paquet (URG, ACK, PSH, RST, SYN, FIN, reportez-vous à la RFC 793 ou à notre article sur le sniffing pour obtenir la signification de chacun de ces champs), le numéro de port sur lequel envoyer le paquet et le nombre de paquets à envoyer.

De suite, vous voyez que nous pouvons envoyer des paquets en spécifiant une adresse source de notre choix (spoofing ?) et ce forgeur se limite à

```
le mode promiscuous");
exit(0);
}

printf("L'interface [%s] est\n",
       "en mode promiscuous",
       interface);
return(0);
}

----- fin sniffer.c -----
```

Vous pouvez tester ce listing en le compilant. Il faut être root pour exécuter ce programme.

```
Raf@garfield > cc -o sniffer sniffer.c
Raf@garfield > ./sniffer rtk0
[...]
```

IPPROTO_TCP); permet de créer une socket.

2 La fonction mode_promiscuous() va activer le mode promiscuous de l'interface réseau.

3 ioctl(sock, SIOCGIFFLAGS, &ifr) récupère la configuration actuelle de l'interface réseau.

4 ifr.ifr_flags |= IFF_PROMISC; active le flag promiscuous.

5 Pour que l'interface prenne en compte les changements, il faut appliquer les modifications avec ioctl(sock, SIOCSIFFLAGS, &ifr).

Désormais, la carte est en mode promiscuous.

Voilà, j'espère que vous avez tout compris et que vous pourrez faire évoluer ce début de sniffer TCP comme vous le voulez. Il pourrait n'afficher que les paquets contenant certains mots-clés par exemple... :-)

Raf

```
{
    &ifr) == -1))
{
    perror("Impossible de récupérer la configuration de l'interface");
    exit(0);
}

printf("L'interface est : %s\n",
       interface);
perror("Recuperation de la configuration de l'interface");

ifr.ifr_flags |= IFF_PROMISC;
if (ioctl(sock, SIOCSIFFLAGS, &ifr) == -1)
{
    perror("Impossible d'activer
```

Les entêtes servent à inclure les définitions de macros, les structures... Il faut les inclure dans cet ordre, ou sinon vous aurez des erreurs de compilation. Les structures IP et TCP servent à récupérer les bits simplement. Dans la fonction main, nous récupérons d'abord l'interface qui nous intéresse. Si vous êtes sous Linux, ce sera « eth0 », par exemple. Vous pouvez trouver leurs noms en tapant la commande ifconfig, qui vous liste les interfaces réseaux.

La structure sockaddr_in contient diverses informations, dont le port et l'adresse internet de l'émetteur du segment.

Revenons sur le passage en mode promiscuous. On suit les étapes suivantes :

```
1 sock = socket(AF_INET, SOCK_RAW,
```

La définition des structures d'entêtes TCP et IP se trouve dans :

```
- /usr/include/netinet/ip.h
- /usr/include/netinet/tcp.h
```

```
typedef struct packet_s
{
    struct iphdr iph;
    struct tcphdr tcph;
} packet_t;
```

À ce stade, il nous faut une autre structure. Si vous avez lu la RFC 793 sur le protocole TCP comme je l'ai signalé précédemment, vous avez vu qu'il faut calculer un checksum pour le protocole TCP et IP. Mais pour le protocole TCP, ce checksum est calculé avec un pseudo header qui contient quelques informations du protocole TCP. Voici ce pseudo header :

```
typedef struct tcpseudohdr_s
{
    unsigned long saddr;
    unsigned long daddr;
    char unuse;
    unsigned char protocol;
    u_short length;
} tcpseudohdr_t;
```

Un p'tit truc pour éviter de faire la gestion d'erreur à chaque malloc... see and learn the power ;)

```
void *xmalloc(int size)
{
    void *ptr;

    ptr = malloc(size);
    if (ptr == NULL) {
        perror("malloc");
        exit(1);
    }
```

```
    }
    return (ptr);
}
```

Le protocole IP ainsi que IP impliquent que certaines valeurs soient remplies. Ce que nous faisons est un forgeur de paquet ; on forge un paquet et l'on n'envoie rien de plus. Donc certaines valeurs de nos headers (IP ET TCP) ne nécessitent aucune cohérence et c'est pour cela que je vais simplement y mettre des valeurs aléatoires. Cette fonction va simplement me permettre d'obtenir un nombre aléatoire pour me servir de graine (man urandom).

```
unsigned long seed(void)
{
    unsigned long ret;
    FILE *fp;

    fp = fopen("/dev/urandom", "r");
    if (!fp) {
        perror("fopen");
        exit(1);
    }
    ret = fread(buf, 4, 1, fp);
    seed = (unsigned long) (*buf);
    fclose(fp);
    return (seed);
}
```

Vous avez certainement entièrement lu la RFC 793 (TCP) et la RFC 791 (IPv4). Eh bien, maintenant il ne vous reste plus qu'à lire la RFC 1071 ! Les protocoles IP et TCP nécessitent le calcul du checksum et si vous aviez lu la RFC 1071 vous auriez certainement trouvé un code source en C de ce fameux checksum

donc vous sélectionnez, vous copiez et vous collez ;)

```
int checksum(void *addr, short count)
{
    /* Compute Internet Checksum for
    "count" bytes beginning at loca
    tion "addr". */
    int checksum;
    register long sum = 0;

    while( count > 1 ) {
        /* This is the inner loop */
        sum += * (unsigned short*) addr++;
        count -= 2;
    }
    /* Add left-over byte, if any */
    if( count > 0 )
        sum += * (unsigned char *) addr;
    /* Fold 32-bit sum to 16 bits */
    while (sum > 16)
        sum = (sum & 0xffff) + (sum >> 16);
    checksum = ~sum;
    return (checksum);
}
```

Ce que l'on fait maintenant, c'est construire notre paquet TCP à partir des arguments venant de la ligne de commande.

Il nous faut l'adresse de destination et l'adresse source, le type du paquet et le port. Vous connaissez désormais les arguments que va prendre votre fonction. On va aussi y ajouter la structure « paquet » et la structure « sockaddr_in ». Cela va nous permettre de construire, récupérer et envoyer notre paquet dans les fonctions respectives.

```
int construct_packet(
    char *dest, char *src, char *type,
    int port, packet_t *p, sockaddr_t *sin)
```

Tout d'abord nous remplissons l'entête IP. Ensuite, nous remplissons l'entête TCP (RFC 793), le pseudo entête TCP et à la fin on calcule le checksum (RFC 1071) avec le pseudo entête TCP... Bingo !

Remarque : Avant chaque remplissage de structure, ce que nous devons faire, c'est mettre ces structures à zéro. Pour cela, on utilise la fonction `memset` (man 2) :

```
memset(&p->iph, 0,
      sizeof(struct iphdr));
p->iph.ihl = 5;
p->iph.version = 4;
p->iph.tot_len =
    htons(sizeof(packet_t));
p->iph.id = 1;
p->iph.ttl = 32;
p->iph.protocol = IPPROTO_TCP;
p->iph.saddr = inet_addr(src);
p->iph.daddr = inet_addr(dest);
p->iph.check = checksum(&p->iph,
    sizeof(struct iphdr));
memset(&p->tcph, 0,
    sizeof(struct tcphdr));
p->tcph.source = htons(1024);
p->tcph.dest = htons(random());
/* random port */
p->tcph.seq = htonl(random());
p->tcph.ack_seq =
    htonl(htonl(random()));
p->tcph.doff = 5;
if (istramp(type, "URG"))
    p->tcph.urg = 1
else if (istramp(type, "ACK"))
    p->tcph.ack = 1;
else if (istramp(type, "PSH"))
    p->tcph.psh = 1;
else if (istramp(type, "RST"))
    p->tcph.rst = 1;
else if (istramp(type, "SYN"))
```

```
sizeof(p), 0,
    (saddr_t*)&sin,
    sizeof(saddr_t));
if (ret == -1)
    perror("sendto");
else
    fprintf(stdout, "[%s] Packet"
        "%s sent to %s\n",
        type, dest);
} /* while */
} /* else */
}

int main(int ac, char **av) {
    char ch;
    int n, p;
    char *dest, *src, *t;

    if (ac < 2)
        fprintf(stderr, "%s\n", USAGE);
    else {
        srandom(seed());
        n = p = 0;
        t = dest = src = NULL;
        while((ch =
            getopt(ac, av,
                "n:d:s:t:p:")) != -1) {
            switch (ch) {
                case 'n':
                    n = atoi(optarg);
                    break;
                case 'd':
                    dest = malloc(strlen(optarg));
                    dest = strncpy(dest, optarg,
                        strlen(optarg));
                    break;
                case 's':
```

Pour qu'un programme s'exécute en C, il faut obligatoirement une fonction appelée « *main* » :

```
int main(int ac, char **av) {
    char ch;
    int n, p;
    char *dest, *src, *t;

    if (ac < 2)
        fprintf(stderr, "%s\n", USAGE);
    else {
        srandom(seed());
        n = p = 0;
        t = dest = src = NULL;
        while((ch =
            getopt(ac, av,
                "n:d:s:t:p:")) != -1) {
            switch (ch) {
                case 'n':
                    n = atoi(optarg);
                    break;
                case 'd':
                    dest = malloc(strlen(optarg));
                    dest = strncpy(dest, optarg,
                        strlen(optarg));
                    break;
                case 's':
```

Dans le domaine de la programmation réseau, vous pouvez vous en sortir avec seulement quelques headers et un peu de théorie réseau. Il faudra exécuter le programme avec les privilèges root. Car ce type de socket nécessite l'utilisation de certaines routines de kernel accessibles seulement depuis root.

Thomas S.

tions que la fenêtre `stdscr` a en mémoire (comme un appel à `move()` etc.) à notre fenêtre physique. Elle permet donc de se mettre à jour par rapport à la fenêtre que l'on modifie en mémoire.

La fonction `move()`

`int move(int y, int x);`

Cette fonction déplace le curseur virtuel à la ligne `y`, avec `x` en abscisse (ou retrait). Elle nécessite l'appel à `refresh()` pour déplacer le curseur physique.

La fonction `printw()`

`int printw(char *fmt, ...);`

Cette fonction est analogue en tous points à la fonction `printf()`. Les options de formatage sont les mêmes que pour la fonction `printf()`. Elle permet d'afficher une chaîne de caractères à la position courante du curseur.

La fonction `mvprintw()`

`int mvprintw(int y, int x, char *fmt, ...);`

Au lieu d'utiliser `move()` pour déplacer le curseur à un endroit donné puis d'écrire une chaîne de caractères par le biais de la fonction `printw`, une alternative a été trouvée. En effet, `mvprintw()` est une fonction deux-en-un qui rassemble `move()` et `printw()`. Elle se déplace à la ligne `y`, à la colonne `x` et écrit la chaîne de caractères donnée en argument. Cette fonction est très utile si l'on ne veut pas se retrouver avec des pages de code inutile, ou plutôt évitable.

La fonction `curs_set()`

`int curs_set(int visibility);`

Cette fonction très simple nous permet de plus ou moins augmenter/réduire la visibilité du curseur. Pour exemple, `curs_set(0);` permet de cacher le curseur.

Les fonctions `echo()` et `noecho()`

`int echo(void);`

`int noecho(void);`

La fonction `echo()` permet d'afficher ce que l'on frappe à l'écran, à l'instar de `noecho()` qui cache les touches frappées. `echo()` et `noecho()` se contrecarrent mutuellement.

La fonction `box()`

`int box(WINDOW *win, chtype verch, chtype horch);`
Pour rappel, `ncurses` est une bibliothèque permettant de coder des interfaces pour nos programmes. Il est alors possible de tracer des bordures, cadres et autres lignes grâce à cette fonction. Généralement, les constantes "ASC_VLINE" et "ASC_HLINE" sont définies sur les systèmes UNIX, conformes à la norme ANSI/POSIX. "ASC_VLINE" correspond à une ligne droite verticale, et "ASC_HLINE" à une ligne droite horizontale. Nous pourrions alors tracer un cadre pour une fenêtre donnée en arg 1 grâce à `box(stdscr, ACS_VLINE, ACS_HLINE);`

La fonction `namps()`

`int namps(int ms);`

Cette fonction permet d'attendre `x` millisecondes. Elle est donc semblable à `sleep()` bien qu'ici l'argument spécifié ne soit pas de l'ordre des secondes mais des millisecondes.

Les fonctions `standout()` et `standend()`

`int standout(void);`

`int standend(void);`

La fonction `standout()` permet de modifier les attributs d'une zone d'affichage en surbrillance. La fonction `standend()` quant à elle rétablit les attributs comme à l'origine.

Note : La fonction `standout()` est équivalente à `attron(A_STANDOUT)`.

La fonction `standend()` est équivalente à

`attrset(A_NORMAL).`

Afficher du texte

Maintenant que nous sommes bien équipés et parés à toute éventualité, lançons-nous dans ce programme...

```
#include <ncurses.h>

/*
 * Name: printw_centre
 * Desc: Affiche un message au
 * milieu de l'écran
 */

void printw_centre(char message[])
{
    /* LINES et COLS sont définis
    dans les headers de la biblio-
    theque ncurses */
    move(LINES/2,
        (COLS-strlen(message))/2);

    printw(message);
    refresh(); // Rend effectifs le
    move et le printw a
    l'écran physique
}

int main()
{
    initscr(); // On cache le cur-
    curs_set(0); // On masque ce que
    noecho(); // l'utilisateur pour-
    // rait frapper
    box(stdscr, ACS_VLINE, ACS_HLINE);
    // On encadre notre fenêtre prin-
    cipale
}
```

```
/* on boucle indéfiniment avec une
attente de 123 millisecondes pour ne
pas utiliser de trop notre CPU ;-)
mais surtout pour avoir cet effet
"guirlande" régulier et visible. */
while(1) {
    stdout();
    printw_centre("Hello world !");
    napms(123);

    standend();
    printw_centre("Hello world !");
    napms(123);
}

/* Fin du programme */
endwin();
return 0;
}
```

```
% gcc hello_world.c \
-o hello_world -lncurses
```

La fonction `printw_centre()` aurait très bien pu se limiter à deux lignes, mais pour une clarté évidente, j'ai utilisé succinctement `move()` et `printw()` en faisant omission de `mvprintw()`.

On se rend compte du résultat... Enjoy ! et pensez à vos yeux, fermez vite ce programme ! ;)

Boîte de dialogue

Passons maintenant à un programme qui pourrait s'avérer utile ;)

Nous allons essayer de coder un programme qui nous demande un login et un password pour lancer un shell. Utile si on lui attribue le bit `suid` et que l'on attribue son appartenance à `root` avec la commande `chown`, on récupérera de ce fait un shell `root`. On pourrait aussi brancher ce pro-

gramme sur un port, avec inetd ou netcat.

Nous allons à cette occasion introduire la notion de couleur grâce à la fonction `start_color`, et aussi la notion de fenêtre dans une fenêtre, grâce à la fonction `subwin()`.



La fonction `subwin()`

`WINDOW *subwin(WINDOW *orig, int nlines, int ncols, int begin_y, int begin_x);`

Cette fonction crée une sous-fenêtre (souvent dans notre fenêtre principale, à savoir "stdscr") de x lignes, x colonnes, et se positionnant aux coordonnées (y ; x).

La fonction `delwin()`

`int delwin(WINDOW *win);`

Cette fonction supprime la fenêtre win en libérant la mémoire qu'elle occupait grâce à `free()`. À noter qu'il faut supprimer les fenêtres sous-jacentes à une fenêtre avant cette dernière.

La fonction `start_color()`

`int start_color(void);`

Cette fonction nous permet d'inclure des couleurs dans nos fenêtres ; très utile à la bibliothèque ncurses.

leurs dans nos fenêtres ; très utile à la bibliothèque ncurses.

La fonction `init_pair()`

`int init_pair(short pair, short f, short b);`

Cette fonction nous permet d'initialiser une paire de couleurs ; elle est utile pour la fonction `wbkgd()`, en deuxième argument. La paire "n" ainsi initialisée peut être invoquée en utilisant `COLOR_PAIR(n)`.

La fonction `wbkgd()`

`int wbkgd(WINDOW *win, chtype ch);`

Cette fonction définit les couleurs du fond d'une fenêtre win et les caractères qu'elle contient.

La fonction `wgetnstr()`

`int wgetnstr(WINDOW *win, char *str, int n);`

Cette fonction lit au maximum n caractères frappés au clavier dans la fenêtre win, ainsi un buffer overflow n'est pas envisageable. Si l'on entre plus de n caractères, la fonction enverra un beep.

Très bien, à présent, let's code ! :-]

```
#include <string.h> /* Pour
strncmp() */
#include <stdlib.h> /* Pour
system() */
#include <ncurses.h>

#define SHELL "/bin/bash"
#define USER "guest"
#define PASS "generic"

void printw_centre(char message[])
{
    mvprintw(LINES/2, (COLS-
        strlen(message))/2,
        message);
    refresh();
}

int main(void)
{
    char buffer_login[10];
    char buffer_pass[10];
    WINDOW *login;
    WINDOW *passd;

    initscr();
    curs_set(0);
    start_color();

    init_pair(1, COLOR_WHITE,
        COLOR_BLUE);

    login = subwin(stdscr, 6, 35, LINES/4,
        (COLS-4)/4);
    wbkgd(login, COLOR_PAIR(1));
    box(login, ACS_VLINE, ACS_HLINE);
    mvprintw(login, 0, 7,
        "Hackademy -- Login ");
    mvprintw(login, 2, 9,
        "Login is: ");
```

```
clear();
printw_centre("Successfully
authenticated, going to have a
prompt...");
```

```
steep(3);

delwin(login);
delwin(passwd);
endwin();

system(SHELL);

return 0;
}
```

Nous avons donc codé un programme qui demande un login, vérifie s'il est égal à la macro USER, puis teste le mot de passe, vérifie s'il est égal à la macro PASS, et si et seulement si ces

deux conditions sont bien remplies, alors notre programme va nous le confirmer au milieu de l'écran (on aura repris la fonction printw_centre) et va nous laisser la main dans un shell (définissable par SHELL). Enjoy !

Ce programme peut bien sûr être utilisé en remote, en le couplant à xinetd ou tout autre programme de ce genre et si on lui ajoute le bit suid (chmod +s login) et qu'il appartient à root, alors le shell délivré sera un shell root.

En écrivant cet article, j'espère vous avoir ouvert un minimum les yeux sur la puissance qu'offre ncurses !

Pour plus d'infos ou pour télécharger la dernière version de cette bibliothèque, je vous invite à visiter cette page web :

<http://www.gnu.org/software/ncurses/biluf>

```
Hackademy -- Passwd
Password is:
[ OK ]
```

LE SITE INTERNET DE THE HACKADEMY

FORUM

CHAT

ANCIENS NUMÉROS FORMATIONS

UNE COMMUNAUTÉ RÉPOND À VOS QUESTIONS

www.thehackademy.net