Faculté des Sciences, AMU — Mercredi 18 décembre 2013 – Durée 2h
Licence d'Informatique — UE SIN5U3 – Sites Luminy et Aix

Réseau et Communication

Vos notes personnelles du cours/TD/TP et les slides du cours sont autorisés ; les autres documents, annales, calculettes et ordinateurs sont interdits.

Vous pouvez appeler sans les recopier les fonctions de la « boîte à outil réseau » vues en TD.

I. Validation de mots de passe en SHA1

On se propose de réaliser en C99 un serveur de validation de mots de passe basé sur le calcul de hash SHA1. Le hash SHA1 est une fonction mathématique non inversible qui permet de caractériser un texte de manière très robuste.

1) Écrire la fonction `int lire_texte (int fd, char *buf, int buf_size)`, qui reçoit en paramètres un descripteur de fichier `fd` ouvert, et un buffer `buf` de taille `buf_size`. La fonction fait autant de `read` que nécessaire pour lire l'intégralité du fichier et le mémoriser dans `buf`. Si le fichier est trop grand il est tronqué (ce n'est pas une erreur). En cas de succès, un zéro terminal est rajouté à `buf` et la fonction renvoie la longueur du texte dans `buf` ; sinon elle affiche un message d'erreur et renvoie -1.

2) Nous allons utiliser la commande `sha1pass` qui prend en arguments un mot de passe et un "grain de sel" (salt). Ce grain de sel est un mot permettant de complexifier le hash, qui dépend donc du mot de passe *et* du grain de sel. La commande affiche sur la sortie standard le hash résultant :

```
<> sha1pass BWV564 JS.Bach
$4$JS.Bach$Fv35Z9qLZr2aYQiyLz6eEL71Xy8$
<> sha1pass BWV564 Toccata
$4$Toccata$/83IlVDjUpA9TaCY4dYFmHvPjFE$
```

Dans cette exemple, le prompt est `<>`, le mot de passe est `"BWV564"`, et les grains de sel sont successivement `"JS.Bach"` et `"Toccata"`.

Écrire la fonction `int calculer_sha1 (const char *pass, const char *salt, char *sha1, int sha1_size)`, qui reçoit en paramètres un mot de passe `pass` et un grain de sel `salt`, calcule leur hash SHA1 et le mémorise dans le buffer `sha1` de longueur `sha1_size`. La fonction crée un tube et un fils ; le fils redirige sa sortie standard sur le tube puis se recouvre avec `sha1pass` ; le père attend la fin du fils, puis lit le texte jusqu'à terminaison dans le tube et le mémorise dans `sha1`. La fonction détecte toutes les erreurs possibles et affiche les messages appropriés. Elle renvoie -1 pour erreur, 0 pour succès.

3) Écrire la fonction `int decomposer_sha1 (const char *sha1, char *salt, char *hash)`, qui reçoit en paramètres un hash SHA1 `sha1` sous la forme `"$4$`*salt*`$`*hash*`$"` (voir exemples dans la question 2), et des buffers `salt` et `hash` de taille suffisante. La fonction extrait le *salt* et le *hash* en se servant de `sscanf`, et les mémorise dans `salt` et `hash`. Elle renvoie 0 pour succès, -1 en cas de mauvaise syntaxe.

4) Écrire la fonction `int valider_pass (const char *sha1, const char *pass)`, qui reçoit en paramètres un hash SHA1 `sha1` et un mot de passe `pass`. La fonction décompose `sha1` en un *salt* et un *hash* ; elle calcule ensuite le hash SHA1 `sha1bis` pour `pass` et *salt*, et enfin compare `sha1` avec `sha1bis` (en se limitant à la taille de `sha1`, car `sha1bis` étant le résultat d'une commande, il peut contenir des retours chariot terminaux). Si cette comparaison réussit, alors `pass` est bien le mot de passe encodé dans `sha1` et la fonction renvoie 0 (succès), sinon renvoie -1.

5) On suppose disposer des fonctions (habituellement vues en TD ; ne pas les recopier !) `int lire_buf (int fd, char *buf, int buf_size)` et `int ecrire_buf (int fd, const char *buf, int buf_len)`. `lire_buf` fait une lecture dans `fd` d'au plus `buf_size-1` caractères, rajoute un zéro terminal en cas de succès et renvoie le résultat de `read`. `ecrire_buf` fait une écriture dans `fd` d'au plus `buf_len` caractères et renvoie le résultat de `write`. Les deux fonctions affichent les messages d'erreur.

Écrire la fonction `int dialoguer_et_valider (int fd0, int fd1)`, qui reçoit en paramètres un descripteur `fd0` ouvert en lecture et un descripteur `fd1` ouvert en écriture (ces deux descripteurs peuvent être identiques dans le cas d'un descripteur bidirectionnel). La fonction implémente le petit protocole suivant : elle fait une lecture dans `fd0` ; la forme attendue étant `"TRY` *sha1 pass*`"`, elle en extrait *sha1* et *pass*, sinon écrit `"SYNTAX ERROR\n"` dans `fd1` ; elle valide ensuite le mot de passe, puis écrit `"OK!\n"` dans `fd1`, sinon écrit `"BAD\n"` dans `fd1`. La fonction renvoie -1 (erreur), 0 (fin de fichier), > 0 (succès). Voir les exemples à la fin.

6) Dans cette question nous allons utiliser la fonction `poll` vue en cours. Cette fonction agit comme `select` et renvoie le même résultat, mais avec un paramétrage différent :

```
#include <poll.h>

int poll (struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
   int   fd;         /* file descriptor */
   short events;     /* requested events */
   short revents;    /* returned events */
};
```

Le `timeout` est en millisecondes (-1 pour désactiver). `fds` est une liste de descripteurs à scruter, de taille `nfds`. Pour tester l'éligilité d'un `fd` en lecture on met `POLLIN` dans `events` avant la scrutation, puis après l'appel on teste si `revents` le contient (avec un *et* binaire).

Écrire la fonction `int scruter_et_dialoguer (int soc_ec)`, qui reçoit en paramètre une socket d'écoute ouverte `soc_ec`. La fonction scrute à l'aide de `poll` l'entrée standard et la socket d'écoute. Si l'entrée standard est éligible, elle dialogue et valide un mot de passe (avec affichage sur la sortie standard). Si la socket d'écoute est éligible, elle accepte la connexion, puis dialogue et valide un mot de passe (avec affichage dans la socket), enfin ferme la connexion. La fonction gère tous les cas d'erreur et affiche les messages appropriés. Elle renvoie 0 pour succès, -1 pour erreur.

En supposant disposer d'un `main` (ne pas l'écrire) qui ouvre une socket d'écoute puis appelle `scruter_et_dialoguer` en boucle, voici un exemple de dialogues :

```
TRY BWV564
SYNTAX ERROR
TRY $4$JS.Bach$Fv35Z9qLZr2aYQiyLz6eEL71Xy8$ BWV565
BAD
TRY $4$JS.Bach$Fv35Z9qLZr2aYQiyLz6eEL71Xy8$ BWV564
OK!
```

Correction

Les fonctions de la « boîte à outil réseau » vues en TD peuvent être appelées sans les recopier.

I. Validation de mots de passe en SHA1

On se propose de réaliser en C99 un serveur de validation de mots de passe basé sur le calcul de hash SHA1. Le hash SHA1 est une fonction mathématique non inversible qui permet de caractériser un texte de manière très robuste.

1) Écrire la fonction `int lire_texte (int fd, char *buf, int buf_size)`, qui reçoit en paramètres un descripteur de fichier `fd` ouvert, et un buffer `buf` de taille `buf_size`. La fonction fait autant de `read` que nécessaire pour lire l'intégralité du fichier et le mémoriser dans `buf`. Si le fichier est trop grand il est tronqué (ce n'est pas une erreur). En cas de succès, un zéro terminal est rajouté à `buf` et la fonction renvoie la longueur du texte dans `buf` ; sinon elle affiche un message d'erreur et renvoie -1.

```
int lire_texte (int fd, char *buf, int buf_size)
{
    int k, pos = 0;
    while (1) {
        k = read (fd, buf+pos, buf_size-1-pos);
        if (k < 0) { perror ("read"); return -1; }
        if (k == 0) break;
        pos += k;
    }
    buf[pos] = '\0';
    return pos;
}
```

2) Nous allons utiliser la commande `sha1pass` qui prend en arguments un mot de passe et un "grain de sel" (salt). Ce grain de sel est un mot permettant de complexifier le hash, qui dépend donc du mot de passe *et* du grain de sel. La commande affiche sur la sortie standard le hash résultant :

```
<> sha1pass BWV564 JS.Bach
$4$JS.Bach$Fv35Z9qLZr2aYQiyLz6eEL71Xy8$
<> sha1pass BWV564 Toccata
$4$Toccata$/83IlVDjUpA9TaCY4dYFmHvPjFE$
```

Dans cette exemple, le prompt est `<>`, le mot de passe est `"BWV564"`, et les grains de sel sont successivement `"JS.Bach"` et `"Toccata"`.

Écrire la fonction `int calculer_sha1 (const char *pass, const char *salt, char *sha1, int sha1_size)`, qui reçoit en paramètres un mot de passe `pass` et un grain de sel `salt`, calcule leur hash SHA1 et le mémorise dans le buffer `sha1` de longueur `sha1_size`. La fonction crée un tube et un fils ; le fils redirige sa sortie standard sur le tube puis se recouvre avec `sha1pass` ; le père attend la fin du fils, puis lit le texte jusqu'à terminaison dans le tube et le mémorise dans `sha1`. La fonction détecte toutes les erreurs possibles et affiche les messages appropriés. Elle renvoie -1 pour erreur, 0 pour succès.

```
int calculer_sha1 (const char *pass, const char *salt, char *sha1, int sha1_size)
{
    int p[2];

    if (pipe (p) < 0) { perror ("pipe"); return -1; }

    int f = fork();
    if (f < 0) { perror ("fork"); close (p[0]); close (p[1]); return -1; }
```

```
    if (f == 0) {
        dup2 (p[1], 1); close (p[1]);
        close (p[0]);
        execlp ("sha1pass", "sha1pass", pass, salt, NULL);
        perror ("exec sha1pass");
        exit (1);
    }

    int status;
    wait (&status);
    if (WEXITSTATUS (status) != 0) {
        fprintf (stderr, "Echec exec\n");
        close (p[0]); close (p[1]); return -1;
    }

    close (p[1]);
    int k = lire_texte (p[0], sha1, sha1_size);
    close (p[0]);
    return k < 0 ? -1 : 0;
}
```

3) Écrire la fonction `int decomposer_sha1 (const char *sha1, char *salt, char *hash)`, qui reçoit en paramètres un hash SHA1 `sha1` sous la forme "4*salt*$*hash*$" (voir exemples dans la question 2), et des buffers `salt` et `hash` de taille suffisante. La fonction extrait le *salt* et le *hash* en se servant de `sscanf`, et les mémorise dans `salt` et `hash`. Elle renvoie 0 pour succès, -1 en cas de mauvaise syntaxe.

```
int decomposer_sha1 (const char *sha1, char *salt, char *hash)
{
    int k = sscanf (sha1, "$4$%[^$]$%[^$]$", salt, hash);
    return k == 2 ? 0 : -1;
}
```

4) Écrire la fonction `int valider_pass (const char *sha1, const char *pass)`, qui reçoit en paramètres un hash SHA1 `sha1` et un mot de passe `pass`. La fonction décompose `sha1` en un *salt* et un *hash* ; elle calcule ensuite le hash SHA1 `sha1bis` pour `pass` et *salt*, et enfin compare `sha1` avec `sha1bis` (en se limitant à la taille de `sha1`, car `sha1bis` étant le résultat d'une commande, il peut contenir des retours chariot terminaux). Si cette comparaison réussit, alors `pass` est bien le mot de passe encodé dans `sha1` et la fonction renvoie 0 (succès), sinon renvoie -1.

```
int valider_pass (const char *sha1, const char *pass)
{
    char hash[80], salt[80], sha1bis[80];

    if (decomposer_sha1 (sha1, salt, hash) < 0) return -1;

    if (calculer_sha1 (pass, salt, sha1bis, sizeof sha1bis) < 0) return -1;

    return strncmp (sha1, sha1bis, strlen(sha1)) == 0 ? 0 : -1;
}
```

5) On suppose disposer des fonctions (habituellement vues en TD ; ne pas les recopier !) `int lire_buf (int fd, char *buf, int buf_size)` et `int ecrire_buf (int fd, const char *buf, int buf_len)`. `lire_buf` fait une lecture dans `fd` d'au plus `buf_size-1` caractères, rajoute un zéro terminal en cas de succès et renvoie le résultat de `read`. `ecrire_buf` fait une écriture dans `fd` d'au plus `buf_len` caractères et renvoie le résultat de `write`. Les deux fonctions affichent les messages d'erreur.

Écrire la fonction `int dialoguer_et_valider (int fd0, int fd1)`, qui reçoit en paramètres un descripteur `fd0` ouvert en lecture et un descripteur `fd1` ouvert en écriture (ces deux descripteurs peuvent être identiques dans le cas d'un descripteur bidirectionnel). La fonction implémente le petit protocole suivant : elle fait une lecture dans `fd0` ; la forme attendue étant `"TRY` *sha1 pass*`"`, elle en extrait *sha1* et *pass*, sinon écrit `"SYNTAX ERROR\n"` dans `fd1` ; elle valide ensuite le mot de passe, puis écrit `"OK!\n"` dans `fd1`, sinon écrit `"BAD\n"` dans `fd1`. La fonction renvoie -1 (erreur), 0 (fin de fichier), > 0 (succès). Voir les exemples à la fin.

```
int dialoguer_et_valider (int fd0, int fd1)
{
    char ligne[100], sha1[100], pass[100]; int k, n, v;

    k = lire_buf (fd0, ligne, sizeof(ligne));
    if (k <= 0) return k;

    n = sscanf (ligne, "TRY %s %s\n", sha1, pass);
    if (n != 2) {
        k = ecrire_buf (fd1, "SYNTAX ERROR\n", 13);
        return k;
    }

    v = valider_pass (sha1, pass);
    k = ecrire_buf (fd1, v < 0 ? "BAD\n" : "OK!\n", 4);
    return k;
}
```

6) Dans cette question nous allons utiliser la fonction `poll` vue en cours. Cette fonction agit comme `select` et renvoie le même résultat, mais avec un paramétrage différent :

```
#include <poll.h>

int poll (struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
   int   fd;         /* file descriptor */
   short events;     /* requested events */
   short revents;    /* returned events */
};
```

Le `timeout` est en millisecondes (-1 pour désactiver). `fds` est une liste de descripteurs à scruter, de taille `nfds`. Pour tester l'éligilité d'un `fd` en lecture on met `POLLIN` dans `events` avant la scrutation, puis après l'appel on teste si `revents` le contient (avec un *et* binaire).

Écrire la fonction `int scruter_et_dialoguer (int soc_ec)`, qui reçoit en paramètre une socket d'écoute ouverte `soc_ec`. La fonction scrute à l'aide de `poll` l'entrée standard et la socket d'écoute. Si l'entrée standard est éligible, elle dialogue et valide un mot de passe (avec affichage sur la sortie standard). Si la socket d'écoute est éligible, elle accepte la connexion, puis dialogue et valide un mot de passe (avec affichage dans la socket), enfin ferme la connexion. La fonction gère tous les cas d'erreur et affiche les messages appropriés. Elle renvoie 0 pour succès, -1 pour erreur.

En supposant disposer d'un `main` (ne pas l'écrire) qui ouvre une socket d'écoute puis appelle `scruter_et_dialoguer` en boucle, voici un exemple de dialogues :

```
TRY BWV564
SYNTAX ERROR
TRY $4$JS.Bach$Fv35Z9qLZr2aYQiyLz6eEL71Xy8$ BWV565
BAD
TRY $4$JS.Bach$Fv35Z9qLZr2aYQiyLz6eEL71Xy8$ BWV564
OK!
```

```
int scruter_et_dialoguer (int soc_ec)
{
    struct pollfd fds[2];

    fds[0].fd = 0;
    fds[0].events = POLLIN;
    fds[1].fd = soc_ec;
    fds[1].events = POLLIN;

    int res = poll (fds, 2, -1);

    if (res < 0) {
        if (errno == EINTR) return 0;
        perror ("poll"); return -1;
    }

    if (fds[0].revents & POLLIN) {
        int k = dialoguer_et_valider (0, 1);
        if (k <= 0) return -1;
    }

    if (fds[1].revents & POLLIN) {
        struct sockaddr_in adr;
        int soc_se = bor_accept_in (soc_ec, &adr);
        if (soc_se < 0) return -1;
        dialoguer_et_valider (soc_se, soc_se);
        close (soc_se);
    }

    return 0;
}
```

Pour compléter la correction, voici les fonctions manquantes ainsi qu'un `main` :

```
#include "bor-util.h"
#include <poll.h>

int ouvrir_port (int port)
{
    struct sockaddr_in adr;

    int soc = socket (AF_INET, SOCK_STREAM, 0);
    if (soc < 0) { perror ("socket ip"); return -1; }

    adr.sin_family = AF_INET;
    adr.sin_port = htons (port);
    adr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bor_bind_in (soc, &adr) == -1) { close (soc); return -1; }
    if (bor_getsockname_in (soc, &adr) < 0) { close (soc); return -1; }
    if (listen (soc, 8) < 0) { perror ("listen"); close (soc); return -1; }

    printf ("port %d ouvert\n", ntohs(adr.sin_port));
    return soc;
}

int lire_buf (int fd, char *buf, int buf_size)
{
    int k = read (fd, buf, buf_size-1);
    if (k < 0) { perror ("read"); return -1; }
    if (k == 0) { printf ("Fermeture fd %d\n", fd); return 0; }
    buf[k] = '\0';
    return k;
}

int ecrire_buf (int fd, const char *buf, int buf_len)
{
```

```
    int k = write (fd, buf, buf_len);
    if (k < 0) { perror ("write"); return -1; }
    return k;
}

int boucle_princ = 1;
void capter_sigint (int sig)
{
    printf ("Serveur: signal %d capté\n", sig);
    boucle_princ = 0;
}

int main (int argc, char *argv[])
{
    int k = 0;

    if (argc-1 != 1) {
        fprintf (stderr, "Usage: %s port\n", argv[0]);
        exit (1);
    }

    int port = atoi (argv[1]);

    bor_signal (SIGPIPE, SIG_IGN, SA_RESTART);
    bor_signal (SIGINT, capter_sigint, 0);

    int soc_ec = ouvrir_port (port);
    if (soc_ec < 0) exit (1);

    while (boucle_princ) {
        k = scruter_et_dialoguer (soc_ec);
        if (k < 0) break;
    }

    close (soc_ec);
    exit (k < 0 ? 1 : 0);
}
```