# Projet de Cryptographie

## Implémentation de l'algorithme Diffie-Hellman dans un logiciel client serveur, et utilisation de Blowfish

OLLIVIER Romain

BASTIDE Vincent

UCBL - Année scolaire 2008 – 2009

# 1 Présentation

Le but de ce projet est de sécuriser l'échange d'informations d'un logiciel client-serveur. Pour cela nous avons choisit d'utiliser l'algorithme Diffie-Hellman, puis le cryptage des informations se fait grâce à Blowfish. Le langage utilisé est le C.

Présentation du logiciel:

Le logiciel client installé sur une machine se connecte au serveur à chaque connexion à Internet. Si une personne perd son ordinateur et que quelqu'un l'utilise pour se connecter à Internet, la personne ayant perdu sa machine pourra via le site web:

-Obtenir un rapport détaillé des dates de connexions ainsi que des IP de sa machine perdue ;
-Récupérer des données stockées sur cette machine;
-Empêcher l'accès à ces données sur la machine une fois qu'elles auront été récupérées (non implémenté).

Le programme utilise les sockets, et donc la suite de protocoles TCP/IP. Actuellement, tous les échanges se font en clair sur le réseau. Le protocole est le suivant:

Identification:
Le client se connecte,
Le serveur lui demande son login (message: «login?»)
Le client répond,
Idem avec le password.

Transferts de fichiers :
Le serveur demande la taille du fichier X («size of X»),
Le client lui répond,
Le serveur demande l'envoie du fichier X («send X»),
Le client envoie le fichier X en plusieurs trames.

Fermeture d'une connexion :
Le serveur envoie le message «close».

Le serveur utilise une base de données afin de stocker toutes les informations relatives aux clients.

# 2 Le travail réalisé

## 2.1 Diffie-Hellman

La méthode d'échange de clés Diffie-Hellman (du nom de ses créateurs Whitfield Diffie et Martin Hellman) concerne les systèmes symétriques. Cette méthode permet à deux personnes Alice et Bob de pouvoir générer une même clé privée qu'ils seront les seuls à connaître, sans que cette clé ne soit diffusée sur le réseau. La clé ainsi généré sera ensuite utilisée par

Blowfish pour crypter les échanges entre Alice et Bob.

### 2.1.1 Algorithme

Voici comment fonctionne cette méthode :

Alice et Bob se sont mis d'accord au préalable sur 2 nombres : Soit un nombre premier **p** assez grand, et une base **g** génératrice des entiers inférieurs à p.
Ils génèrent tous deux de leur coté un nombre aléatoire, **a** pour Alice, **b** pour Bob.

Etape 1:
Alice calcule $A = g^a \pmod p$ et l'envoie à Bob.
Bob calcule $B = g^b \pmod p$ et l'envoie à Alice.

Etape 2:
Alice calcule $k = B^a \pmod p$.
Bob calcule $k' = A^b \pmod p$.

Mathématiquement, on remarque que **k** = **k'**. Alice et Bob on ainsi pu générer la même clef qu'ils sont les seuls à connaître.
Si quelqu'un avait pu intercepter **A** et **B** à l'étape 1, il n'aurai pas pu en déduire **k**. Cela vient du fait que la fonction modulo n'est pas réversible. En effet, si y = x modulo (constante), on peut trouver y à partir de x, mais il existe une infinité de solution si on veut trouver x à partir de y :
Soit x = 20, constante = 13, 20 modulo 13 = 7,
mais on a aussi 7 = 33 modulo 13 = 46 modulo 13, etc...

On notera également l'importance de l'ordre de grandeur de p, qui définit directement les bornes de la clef k qui sera comprise entre 1 et p-1.

### 2.1.2 Travail réalisé

La principale difficulté pour l'implémentation des fonctions permettant la génération des clés est de pouvoir gérer des nombres de très grande taille. Dans notre cas, le nombre premier déterminé au préalable est généralement composé de plus de 100 chiffres. Les types de base du C ne permettent pas de stocker et d'opérer sur de tels nombres.

Nous avons donc utilisé la bibliothèque libre GMP (GNU Multiprecision Library) qui fourni un ensemble de types permettant la manipulation de nombres (entiers, réels, etc) dont la taille n'est limité que par la mémoire de la machine sur laquelle elle est utilisée. Elle fournit également l'équivalent des méthodes de math.h, à savoir pow(), modulo, etc.

La base **g** et le nombre premier **p** sont déterminés à l'avance et définis dans le code.

Nous avons implémenté les fonctions permettant de générer les clés, que voici :

```
void genererNbAleatoire(mpz_t nb) ;
```

```
void genererDH(mpz_t a,mpz_t DH) ;
void calculerCle(mpz_t a, mpz_t infoB, mpz_t cle) ;
char * cleToChar(mpz_t cle, int base) ;
void charToCle(char * chaine, mpz_t res, int base) ;
```

- genererNbAleatoire() permet tout d'abord d'obtenir un entier aléatoire, compris entre 1 et **p-1** dans notre cas. Ce nombre est celui choisi par Alice et Bob au tout début du processus. Il est de type mpz_t, le type entier de la libraire GMP.

- GenererDH() est la fonction qui va déterminer l'information que chaque interlocuteur enverra à l'autre, à savoir le nombre **g** élevé à la puissance du nombre aléatoire, modulo du nombre premier **p**.

- CalculerCle() va calculer la clé qui sera utilisée pour le cryptage, et la stockera dans un mpz_t.

Les fonctions cleToChar() et charToCle() permettent la conversion d'un mpz_t vers une chaine de caractères et vice-versa. Elles sont utiles notamment pour le transfert des grands nombres sur le réseau.

## 2.2 Blowfish

Blowfish est un algorithme de chiffrement symétrique par blocs comme DES ou IDEA. Il gère des clés de chiffrement allant de 32 bits a 448 bits.

### 2.2.1 Fonctionnement

Blowfish utilise une combinaison de la construction par réseau de Feistel, de la permutation par P-Boxe et de la substitution par S-Boxe.

#### 2.2.1.1 Le Réseau de Feistel

Le réseau de Feistel est une manière de construire des blocs. Il est subdivisé en «tours» (16 dans le cas de blowfish). Le principe est le suivant:
Un bloc de x bits est divisé en deux blocs de x/2 bits.
A chaque tour, un bloc est chiffré avec une sous-clé (explications dans la suite), puis est combiné à l'autre bloc bits à bits avec l'opération XOR (ou exclusif).
Au tour suivant, les blocs sont inversés.

Exemple: Soient deux blocs A et B, B est chiffré avec la première sous-clé en B', puis est combiné à A (A xor B'), A devient alors A'. Au tour suivant, on répétera le même algorithme en inversant A' et B' avec une nouvelle sous-clé.

Le déchiffrement se passe dans le sens inverse, et doit par conséquent comporter le même nombre de tours, et les même sous-clés.

Les sous-clés sont calculées en fonction du nombre de tours du réseau de Feistel ainsi que de la clé principale. Dans le cas de blowfish, voir le paragraphe 2.2.1.3 Algorithme pour voir comment elles sont générées.

### 2.2.1.2 Les S-Boxes et P-Boxes

Les S-Boxes sont des tables de substitutions à deux dimensions, c'est à dire qu'une suite binaire donnée à la S-Boxe est remplacée par une autre suite binaire. Les suites de bits en entrée et sorties ne font pas forcément la même taille. Dans le cas de Blowfish, ces S-Boxes sont générées en fonction de la clé, nous verrons dans le paragraphe suivant comment.

Les P-Boxes sont des tableaux à une dimension utilisées pour permuter des éléments d'une structure.

### 2.2.1.3 Algorithme

P-array (ou tableau P): C'est un tableau à une dimension qui contient 18 chiffres hexadécimaux qui correspondent aux chiffres de π (sans compter le 3).

Exemple: π-3 = 0.141592653589793, ce qui donne en hexadécimal 243F6A88 – 85A3..., donc le premier élément de P est P1 = 243F6A88 (voir blowfish.c du projet), le deuxième est P2 = 85A308D3, etc..

La S-Boxe de blowfish est un tableau à deux dimensions (4 par 256) contenant également des chiffres de π sous forme hexadécimale. Par le suite, on définira Si,j la case de S correspondante à la ligne i et colonne j, i allant de 1 à 4 et j de 1 à 256.

Pourquoi π?

Le choix de π pour remplir les tableaux précédents ne vient pas d'une raison mathématique. π a été choisi pour montrer qu'aucune décision arbitraire n'a été prise afin de remplir ces tableaux, et par conséquent la procédure de chiffrement peut-être vérifiée par n'importe qui, et universellement (car π est connu de tous).

En effet, la cryptographie se base sur un principe fondamental (voir principes de Kerckhoffs): l'algorithme utilisé doit être connu, et cela ne doit pas avoir d'impact sur la facilité ou non à le casser. Une information chiffrée ne doit pouvoir être déchiffrée qu'en connaissant la clé secrète. Ainsi, un algorithme peut-être soumis à des tests de la communauté, validant ainsi sa robustesse. C'est d'ailleurs ainsi qu'à procéder Bruce Schneier en 1995 dans un article paru du journal « Dr Dobb's journal », disant qu'un an après sa publication, blowfish n'avait toujours pas été cassé.

Revenons à Blowfish. Avant de chiffrer des données, les tableaux P et S doivent être initialisés en fonction de la clé d'une façon complexe. Nous ne sommes pas entrés dans les détails mais on retient que chaque case des tableaux est soumise à un XOR avec 32 bits de la clé puis cryptée récursivement avec l'algorithme de blowfish.

A la fin de cette opération, P et S sont donc initialisés de façon correcte, et les données peuvent être chiffrées avec l'algorithme blowfish qui est le suivant:

Soit x un bloc de 64 bits (qui est un morceau de la chaîne à crypter), coupé en deux blocs de 32 bits xL et xR (correspondants aux parties gauches et droites de l'image 1).

Le réseau de Feistel utilisé est le suivant:

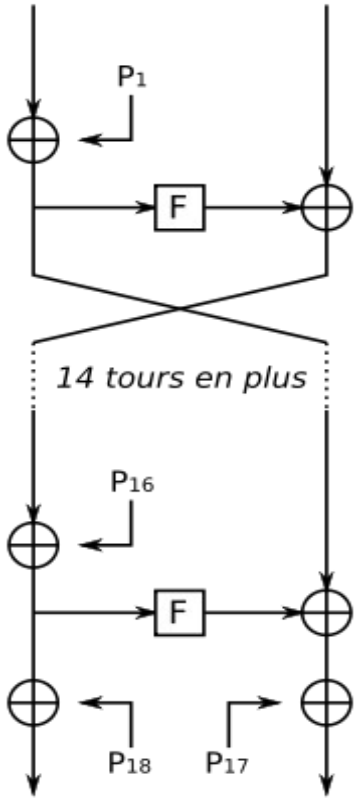


*Image 1: Illustration de l'Algorithme Blowfish*

Pour i allant de 0 à 16 (car 16 tours)

xL = xL XOR Pi
xR = F(xL) XOR xR
swap xL, xR

fin boucle.
Swap xL and xR
xR = xR XOR P17
xL = xL XOR P18 (d'où le besoin de 16 + 2 cases dans P-array)

Puis x = xL concaténé à xR.
On fait donc cela pour chaque bloc de 64bits de la chaîne à chiffrer.
Pour le déchiffrage, le principe est le même, sauf que l'on utilise les éléments de P dans l'ordre inverse.

La fonction F est la suivante:

xL (ou xR) est composé de 4 blocs de 8 bits (soit un char): a,b,c et d, donc chaque bloc allant de 0 à 255.

F(xL) = [ ( S1,a + S2,b mod 2^32 ) XOR S3,c ] + S4,d mod 32

On comprend ici l'utilité d'avoir une S-Box de 4x256:

4 opérations sur 1 octet, soit de 0 à 255 possibilités.

### 2.2.2 Travail réalisé

Au début nous avons utilisé mcrypt, une bibliothèque disponible sur les dépôts officiels. Mais nous avons constaté que sur le site officiel du créateur de blowfish on pouvait trouver des sources utilisables en C, et même 3 différentes. Nous avons choisit celles de Paul Kaucher.

Elles fournissent des fonctions permettant d'initialiser le P-array et la S-Box en fonction de la clé, ainsi que deux fonctions permettant de crypter et décrypter deux blocs de 32 bits correspondants à xL et xR dans les explications précédentes.

```
typedef struct
{
  unsigned long P[16 + 2]; //P-array
  unsigned long S[4][256]; //S-Boxe
} BLOWFISH_CTX;

void Blowfish_Init(BLOWFISH_CTX *ctx, unsigned char *key, int keyLen);
void Blowfish_Encrypt(BLOWFISH_CTX *ctx, unsigned long *xl, unsigned long *xr);
void Blowfish_Decrypt(BLOWFISH_CTX *ctx, unsigned long *xl, unsigned long *xr);
```

On constate bien ici qu'une fois les tableaux P et S sont initialisés la clé n'est plus utilisée dans le reste de l'algorithme, ce qui coïncide avec ce qui a été dit précédement.

Afin d'utiliser ces sources dans le projet, nous avons créé une structure permettant de manipuler ces fonctions sans se soucier de l'algorithme de chiffrement utilisé. Ainsi, si un jour nous décidons de changer d'algorithme, seule cette structure sera à modifier, et non tout le code du projet.

La structure est la suivante (définie dans le fichier crypt.h):

```
typedef struct
{
  BLOWFISH_CTX ctx; //structure des sources provenant du site
} CRYPT;

void initCrypt(CRYPT *_crypt, char *key, int keyLen);
char *Encrypt(char *mess, int *size, CRYPT *_crypt);
char *Decrypt(char *mess, int *size, CRYPT *_crypt);
```

Des fonctions supplémentaires on dues être créées afin de pouvoir manipuler plus facilement une chaîne de caractères reçues (pour la séparer en blocs de 8 octets, puis en deux blocs de 32 bits, puis les convertir en long, et vice versa). En effet, les fonctions fournies dans les sources prennent en paramètre des variables de type unsigned long, ce qui équivaut à 4 octets, soient 32 bits (taille des blocs nécessaire).

Après avoir fait des tests, la structure a été implémentée avec succès dans le projet principal. De nouvelles fonctions se sont avérées nécessaires, ainsi que la modification de fonctions déjà présentes à cause de nouveaux problèmes rencontrés, notamment au niveau de la taille des messages qui devait varier: par exemple, une chaîne de caractères doit avoir une taille divisible par 64 bits puisque blowfish demande absolument des blocs de 64 bits. Il a donc fallu adapter le code afin de toujours être sure de la taille des messages (cryptés ou non).

Après cela, nous avons ajouté encore une petite couche de sécurité au projet afin de rendre la tâche de l'attaquant plus difficile. Au lieu de séparer une chaîne de 8 octets (soit 8 caractères) en deux chaînes de 4 caractères, l'une contenant les 4 premiers (xL) , et l'autre les 4 derniers (xR), nous avons séparés les caractères de façon à ce que xL contienne les numéros paires de la chaîne initiale, et xR les numéros impaires:

Exemple:
La chaîne x : a b c d e f g h serai normalement séparée en xL : a b c d et xR : e f g h
Mais dans notre cas, xL : a c e g et xR : b d f h.

Bien sur cet algorithme n'a rien de robuste et est facilement cassable, mais il a exactement le même coût (au sens complexité) que la méthode de séparation initiale, alors pourquoi s'en priver? De plus, dans le cas d'une attaque known-plaintext, cela peut-être utile.

Afin que la partie chiffrement puisse être mieux constatée, le code envoyé avec ce rapport contient une version modifiée et simplifiée du serveur (pas de connexion aux bases de données). Ainsi, à la fin d'un échange classique entre le client et le serveur (échange Diffie-Hellman, puis login, puis password) le programme server proposera d'envoyer des messages au client.

# 3 Conclusions

Sur Blowfish: A l'heure actuelle, Blowfish est considéré comme plus sure et plus efficace que DES, et n'a pas encore été cassé dans sa version à 16 tours (jamais cassée au-dela de 4 tours) et la seule attaque possible connue reste donc le bruteforce. Ainsi, en utilisant Diffie-Hellman pour générer des clés, l'attaquant aura du mal à deviner les clé générées (vu qu'en général dans le cas d'attaques bruteforces on essai de deviner les mots de passes plutôt que tous les essayer). Blowfish est donc une solution de chiffrement sure et rapide.

Pour Diffie Hellman, la faiblesse réside dans l'attaque dite de l'homme du milieu. On suppose que l'attaquant connaît **g** (et p). Si celui ci peut intercepter et modifier tous les messages

envoyés par Alice et Bob, il peut procéder ainsi :

Il réceptionne g^a envoyé par Alice et envoi g^a' à Bob.

Fait de même en envoyant g^b' à Alice.

Il pourra ensuite communiquer avec Alice avec g^ab' et avec Bob avec g^a'b.

Pour éviter cela, les messages échangés entre Alice et Bob doivent être marqués, d'une signature par exemple.

## Références

[RAM08] Raymond, « Introduction à la cryptographie », ram-0000.developpez.com/tutoriels/cryptographie/, janvier 2008.

[GUI02] S. Guillem-Lessard, «Cryptographie tutoriel», www.uqtr.ca/~delisle/Crypto/ , 2001-2002.

[RSA] RSA.com, « Crypto FAQ », www.rsa.com/rsalabs/node.asp?id=2248

[WIKI] « Blowfish », http://fr.wikipedia.org/wiki/Blowfish, 2009.

[B.SCH] Bruce Scheiner, « Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish) », http://www.schneier.com/paper-blowfish-fse.html

[P.KAU] « Blowfish source code », http://www.schneier.com/blowfish-download.html (sources de Paul Kocher).

[WIKI] « Echange de clés Diffie-Hellman » http://fr.wikipedia.org/wiki/%C3%89change_de_cl%C3%A9s_Diffie-Hellman

[GMP] « Gnu Multiprecision Library » http://gmplib.org/

[PRI] « The Primes Pages » http://primes.utm.edu/