Université de Toulon — Langage C
Licence 2 Informatique — Année 2020-2021

Projet : Échange de clés Diffie-Hellman

1 Présentation

L'échange de clés Diffie-Hellman est une méthode par laquelle des entités peuvent se mettre d'accord sur un nombre sans qu'une autre entité (malveillante) puisse découvrir le nombre, même en ayant écouté tous leurs échanges. Ce nombre échangé pourra ensuite être utilisé comme clé pour chiffrer une conversation.

Article bien détaillé :
`https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange`

2 Objectif

L'objectf ici est d'implémenter le protocole d'échange de clés Diffie-Hellman avec un nombre quelconque d'entités.

3 Le protocole

3.1 Exemple avec deux entités

Supposons que deux entités Alice et Bob souhaitent échanger un sécret avec ce protocole.
Soit p un nombre premier et g un générateur du groupe multiplicatif $((\mathbb{Z}/p\mathbb{Z})^*, \times)$.
Remarque : $((\mathbb{Z}/p\mathbb{Z})^*, \times)$ est un groupe cyclique d'ordre $p-1$. g est appelé une racine primitive modulo p.
Les paramètres p et g sont publics.
Le protocole :

1. Alice génère aléatoirement un nombre a dans $\mathbb{Z}/p\mathbb{Z}$.
2. Alice calcule $A = g^a \pmod{p}$ et l'envoie publiquement à Bob.
3. Bob génère aléatoirement un nombre b dans $\mathbb{Z}/p\mathbb{Z}$.

4. Bob calcule $B = g^b \pmod{p}$ et l'envoie publiquement à Alice.
5. Alice calcule $S_a = B^a \pmod{p}$.
6. Bob calcule $S_b = A^b \pmod{p}$.

Le secret partagé est : $S = S_a = S_b = g^{a*b} \pmod{p}$.

3.2 Généralisation

La généralisation du protocole à un nombre quelconque d'entités est bien détaillée ici :

`https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange#Operation_with_more_than_two_parties`

4 À propos de l'exponentiation modulaire

Soient g, x et p trois entiers. Le calcul $g^x \pmod{p}$ est une exponentiation modulaire. Lorsque les entiers manipulés sont suffisamment grands, l'approche naïve devient impossible en temps et en espace car implique trop un nombre trop élévé d'itérations et des produits temporaires trop grands. Une méhtode efficace pour effectuer cette opération est l'exponentiation modulaire binaire (aussi appelé *square-and-multiply*). Cette méthode utilise la représentation binaire de l'exposent pour accélérer de manière considerable le calcul $g^x \pmod{p}$.
Liens détaillants assez bien cette méthode :

— `https://fr.wikipedia.org/wiki/Exponentiation_modulaire`
— `https://en.wikipedia.org/wiki/Exponentiation_by_squaring`

5 À propos des grands entiers

Pour que ce protocole soit sûr, il est indispensable d'utiliser de grands entiers pour les opérations arithmétiques. Par défaut, les machines classiques permettent en général de manipuler des entiers de 64 bits au maximum. Cette taille est largement inférieure à celles utilisées en cryptographie à clé publique qui varient de 256 à des milliers de bits. Par conséquent, il est donc nécessaire d'avoir recours à des librairies pour la manipulation de grands entiers.

L'une des librairies les plus populaires et performantes dans ce domaine est la librairie GNU MP : `https://gmplib.org/`. Cette librairie permet de manipuler de grands nombres (pas seulemnt les entiers). Ici, nous ne nous intéresserons qu'à la manipulation des grands entiers. Voici quelques liens intéressants vers sa documention :

1. `https://gmplib.org/manual/` : la page principale de la documentation.
2. `https://gmplib.org/manual/GMP-Basics.html` : pour les connaissances générales de base sur la librairie.
3. `https://gmplib.org/manual/Integer-Functions.html` : pour la manipulation de grands entiers.
4. `https://gmplib.org/manual/Formatted-Input.html` : pour la gestion des entrées.
5. `https://gmplib.org/manual/Formatted-Output.html` : pour la gestion des sorties.
6. `https://gmplib.org/manual/Random-Number-Functions.html` : pour la génération de nombres aléatoires.

Ci-dessous, un exemple d'utilisation de la librarie. Ce code génère un nombre premier p de n bits, ensuite génère aléatoirement deux entiers a et b dans $\mathbb{Z}/p\mathbb{Z}$.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <gmp.h>
5
6
7  int main(void){
8
9    int n;
10   unsigned long seed;
11   mpz_t a, b, c, p;
12   gmp_randstate_t r;
13
14   mpz_inits (a, b, c, p, NULL);
15
16   seed = time(NULL);
17   gmp_randinit_default(r);
18   gmp_randseed_ui(r, seed);
19
20   n = 256;
21   // generation d'un nombre premier de n bits
22   do{
23     mpz_urandomb (p, r, n);
24     mpz_setbit (p, n-1);
25     mpz_nextprime (p, p);
```

```
26    }while (mpz_sizeinbase (p, 2) != n);
27
28    // generation de deux nombres aleatoires inferieurs a p
29    mpz_urandomm (a, r, p);
30    mpz_urandomm (b, r, p);
31    gmp_printf("\na = %Zd\n\n", a);
32    gmp_printf("b = %Zd\n\n", b);
33    gmp_printf("p = %Zd\n\n", p);
34
35    // addition
36    mpz_add (c, a, b);
37    gmp_printf("addition, c = %Zd\n\n", c);
38
39    // soustraction
40    mpz_sub (c, a, b);
41    gmp_printf("soustraction, c = %Zd\n\n", c);
42
43    // multiplication puis reduction modulaire
44    mpz_mul (c, a, b);
45    mpz_mod (c, c, p);
46    gmp_printf("multiplication modulaire, c = %Zd\n\n", c);
47
48    // exponentiation modulaire
49    mpz_powm (c, a, b, p);
50    gmp_printf("exponentiation modulaire, c = %Zd\n\n", c);
51
52    mpz_clears (a, b, c, p, NULL);
53    gmp_randclear(r);
54
55    return 0;
56 }
```

Remarques :

1. Pour utiliser la librairie GNU MP, il faut l'avoir installé auparavant ; elle est normalement déjà installée sur les machines de TP.
   Lien pour le téléchargement : `https://gmplib.org/#DOWNLOAD`
   Lien pour l'installation : `https://gmplib.org/manual/Installing-GMP.html`
2. Compilation : l'utilisation de cette librairie nécessite l'ajout de l'option 'lgmp' à la commande de compilation. Ainsi, si le code ci-dessus est enrégistré dans le fichier

'exemple.c', il pourra être compilé avec la commande :
gcc -Wall exemple.c -o exemple -lgmp

6 Taches à effectuer

Pour ce projet, on utilisera la librairie GNU MP pour la manipulation des grands entiers. Les taches à effectuer sont les suivantes :

1. Implémenter l'exponentiation modulaire comme expliquée dans la section 4.

2. Écrire une fonction qui prend en paramètre un entier n et génère aléatoirement un nombre premier p de n bits ainsi qu'un élément g dans $(\mathbb{Z}/p\mathbb{Z})\backslash\{0, 1, p-1\}$.
   Remarque : Comme expliqué plus haut, g devrait être une racine primitive modulo p. Pour ce projet, on se contentera d'un élément g différent de 0, 1 et $p-1$.

3. Écrire une fonction qui simule le protocole d'échange de clés Diffie-Hellman entre un nombre m d'entités. m devra faire partie des paramètres de la fonction. Cette fonction devra également prendre en paramètres les éléments p et g générés au préalable. Ensuite, elle devra produire un secret de taille n bits.
   Enfin, il faudra simuler l'échange de données entre les différentes entités avec des messages. Par exemple : Entité_1 envoie X à Entité_2.

4. Ecrire une fonction (la fonction *main*) qui :

   — Demande à l'utilisateur la taille n en bits du secret à partager ainsi que le nombre m d'entités qui seront impliquées dans le protocole.

   — Exécute le protocole d'échange de clés Diffie-Hellman avec ces entités en utilisant les fonctions précédentes.

   — Vérifie que le résultat est correct et que les différentes entités ont au final le même sécret.