

Atelier n° 1:

Avec clé privée:

Clé aléatoire.

Identique pour encodage et décodage,

Utilisation unique (One Time Pass).

Clé de même longueur que le message.

Un théorème de Claude Shannon (Ingénieur et mathématicien) prouve (en gros et) en 1949 :

Qu'un message chiffré avec une clé de même longueur, aléatoire et utilisée qu'une fois est indéchiffrable par quiconque ne connaissant pas la clé.

<https://www.lavachequicode.fr/cryptographie/cle-a-usage-unique>

http://www.acrypta.com/telechargements/fichecrypto_300.pdf

Méthode de codage par un OU exclusif (XOR) :

Un message est une suite de bits.

Ex: "AAAA" est égal à en ascii à 0100.0001|0100.0001|0100.0001|0100.0001

Une clé aussi

Ex: 0101.1001|1111.0110|0100.0010|0110.0100 ("YöBd")

M= 0100.0001|0100.0001|0100.0001|0100.0001

XOR

K= 0101.1001|1111.0110|0100.0010|0110.0100

donne

C= 0001.1000 ...etc

Puis

C= 0001.1000

XOR

K= 0101.1001

donne:

M= 0100.0001

On va grouper les octets de la clé et du message clair par **4** et stocker la valeur numérique de ces 4 octets dans un tableau.

Pourquoi ? Parce que en JS l'opérateur XOR (^) s'opère sur 32 bits max. $32=8 \times 4$.

Pour ça on fera tous les 4 caractères (de gauche à droite) :

$C1 \times 256^3 + C2 \times 256^2 + C3 \times 256^1 + C4 \Rightarrow$ 1 élément du tableau

ou

$C1 \times 256^3 + C2 \times 256^2 + C3 \times 256^1 + C4 \times 256^0$

Ce principe est identique au calcul du nombre de secondes dans

3h 42mn 22 sec, soit :

$3 \times 60 \times 60 + 42 \times 60 + 22 = 10800 + 2520 + 22 = 13342$

ou

$3 \times 60^2 + 42 \times 60^1 + 22 \times 60^0 = 13342$

Inconvénient : Le transfert sécurisé des clés secrètes (une par message)

Pratique :

Création de la clé aléatoire :

Note :

Dans le prog. principal, on a

```
var o=createKey (plainTxt.length);
```

```
var key=o.key;
```

On écrit donc la fonction :

```
/**
 * Random key creation
 * @param len                Length of plain text
 * @return object with
 *                          key    Created key
 *                          binKey  binary string of key (only to display it)
 */
function createKey (len) {
    var key="";var binKey8=""; var binKey=""; var splitter=".";
    for (i=0;i<len*8;i++) {
        var bit=random();
        binKey8+=""+bit; // binKey8+= String(bit);
        if ( i%8==7) {
            if (i==(len*8)-1) splitter="";
            var c = parseInt(binKey8, 2);
            key+=String.fromCharCode(c);
            binKey+=binKey8+splitter;
            binKey8="";
        }
    }
    //log("len key="+key.length);
    //log("len binStr="+binKey.length);
    return {binKey:binKey , key:key};
}
```

```
// ...
```

```
// Random utililty function
```

```
function random () {
    return Math.floor(Math.random() * 2);
};
```

Notes :

Dans le prog. principal, après

```
var o=createKey (plainTxt.length);
```

```
var key=o.key;
```

On a dans key une clé de même longueur que la message (var plainText)

Regroupement de 4 octets par une valeur de 32bits et rangement dans un tableau (Array).

Note :

Dans le prog. principal, on a

```
// Convert plain text to array of 4 bytes elements
```

```
var arrBy= toArrayOfBy(plainTxt,by);
```

et

```
// Convert key to array of 4 bytes elements
```

```
var keyBy=toArrayOfBy(key,by);
```

On écrit donc le corps de la fonction :

```
/**
 * Creation of an array of 4 bytes (32 bits) elements
 * @param str          A text string (message or key)
 * @param by           Number of bytes (by=4)
 * @return An array of 32 bits elements
 */
function toArrayOfBy (str,by) {
    var arrOut=[]; var c=0;
    for (i=0;i<str.length;i++) {
        c=c*256+str.substr(i,1).charCodeAt(0) ;
        if (i%by==(by-1)) {
            arrOut.push(c);    c=0;
        }
    }
    if (c!=0) arrOut.push(c);
    return arrOut;
}
```

Notes :

Dans le prog. principal, après

```
var arrBy= toArrayOfBy(plainTxt,by);
```

et

```
var keyBy=toArrayOfBy(key,by);
```

On a 2 tableaux de même longueur contenant chacun des valeurs de 32 bits.

Chiffrement (encodage)

Note :

Dans le prog. principal, on a

```
// Ciphering
```

```
var cipherArr=cipher (arrBy,keyBy);
```

On écrit donc le corps de la fonction :

```
/**
 * Ciphering
 * @param msgArr           Message's 32 bits array
 * @param keyArr           Key's 32 bits array
 * @return Ciphered message's 32 bits array
 */
function cipher (msgArr,keyArr) {
    var arrOut=[]; var j=0;
    for (var i in msgArr) {
        if (j>keyArr.length-1) j=0 ;
        arrOut.push((msgArr[i] ^ keyArr[j]));
        j++;
    }
    return arrOut;
}
```

Déchiffrement

Note :

Dans le prog. principal, on a

```
// Ciphering
```

```
var cipherArr=cipher (arrBy,keyBy);
```

On écrit donc le corps de la fonction :

```
/**
 * Unciphering
 * @param cipherArr      Message's 32 bits array
 * @param keyArr          Key's 32 bits array
 * @return Ciphered message's 32 bits array
 */
function uncipher (cipherArr,keyArr) {
    var arrOut=[]; var j=0;
    for (var i in cipherArr) {
        if (j>keyArr.length-1) j=0 ;
        arrOut.push((cipherArr[i] ^ keyArr[j]));
        //log("cipherArr[i]="+cipherArr[i]+" / keyArr[j]="+keyArr[j]);
        j++;
    }
    return arrOut;
}
```

Conversion de tableau déchiffré en texte

Note :

Dans le prog. principal, on a

```
// Uncipher array to string
```

```
var unciphText=arrayByToString (uncipherArr,by)
```

On écrit donc le corps de la fonction :

```
/**
 * Reconstruction of plain text
 * @param arr          Unciphered array
 * @param by           Number of bytes (by=4)
 * @return Original text reconstituted
 */
function arrayByToString (arr,by) {
    var str=""; var strBy=""; var c;
    for (var i=0;i<arr.length;i++) {
        var nBy=arr[i];
        for (var j=0;j<by;j++) {
            c=nBy%256;
            if (c!=0) {
                strBy=String.fromCharCode(c)+strBy;
                nBy=Math.floor(nBy/256);
            }
        }
        str+=strBy;strBy="";
    }
    return str;
}
```

*** ON TESTE ***

Atelier n° 2 : RSA (Rivest, Shamir et Adleman).

Chiffrement avec clé publique / déchiffrement avec clé privée:

Clés construites par le destinataire qui donne la clé publique à (aux) expéditeur(s).

Seule la clé privée permet le décodage.

Utilisation de grand nombre pour construire les clés (>256 ou 512 bits).

RSA est basé sur le fait qu'il est très difficile de factoriser en 2 nombres 1ers, un très grand nombre entier très grand. Même avec un ordinateur le calcul prend un temps "exponentiellement" long.

Ex : A partir de 15 il est facile de dire que $5 \times 3 = 15$ mais donner moi les facteurs de 221 (17×13). Avec 283.189 (503×563)

Inconvénients :

Le temps de calcul pour retrouver la clé privée doit être supérieur au le temps durant lequel le secret doit être conservé.

Nous ne sommes pas à l'abri d'un(e) mathématicien(e) "génial(e)" trouvant un algorithme rapide de factorisation.

Théorie :

Création de la clé publique :

Choix de 2 nombre premiers p et q. \Rightarrow p=503 et q=563

Calcul de $n = p \times q$ $\Rightarrow 503 \times 563$ n=283 189

Calcul d'un nombre $\phi = (p-1 \times q-1)$ $\Rightarrow 502 \times 562$ $\phi=282\ 124$

et d'un nombre e qui doit être

1/ > à p et q

2/ être premier par rapport à ϕ

soit $(PGCD(e, \phi) = 1)$ \Rightarrow e=565

nb : Il faut un algo. pour le calculer.

On a donc la clé publique (e,n) \Rightarrow e=565 et n=283 189

Variables de départ et calculs intermédiaires

p=503 et q=563

n=283 189

$\phi=282\ 124$

Clé publique

e=565 et n=283 189

Pratique :

Note :

Dans le prog. principal, on a

```
var p = 503 ;
```

```
var q = 563 ;
```

```
var n = p*q ;
```

```
var phi= phiOf (p,q) ;
```

```
var e=primeOf(phi,p,q);
```

```
var by=2 ;    // On va regrouper les caractères du message clair par 2
```

```
              // pour ne pas avoir le même code qui se répète)
```

```
var len16=5; // long de la string hexa ( $16^{\text{len16}} > n$ ) (i.e.  $16^5 = 1,048,576 > n$ )
```

```
              // cette string ne sert que pour la transmission.
```

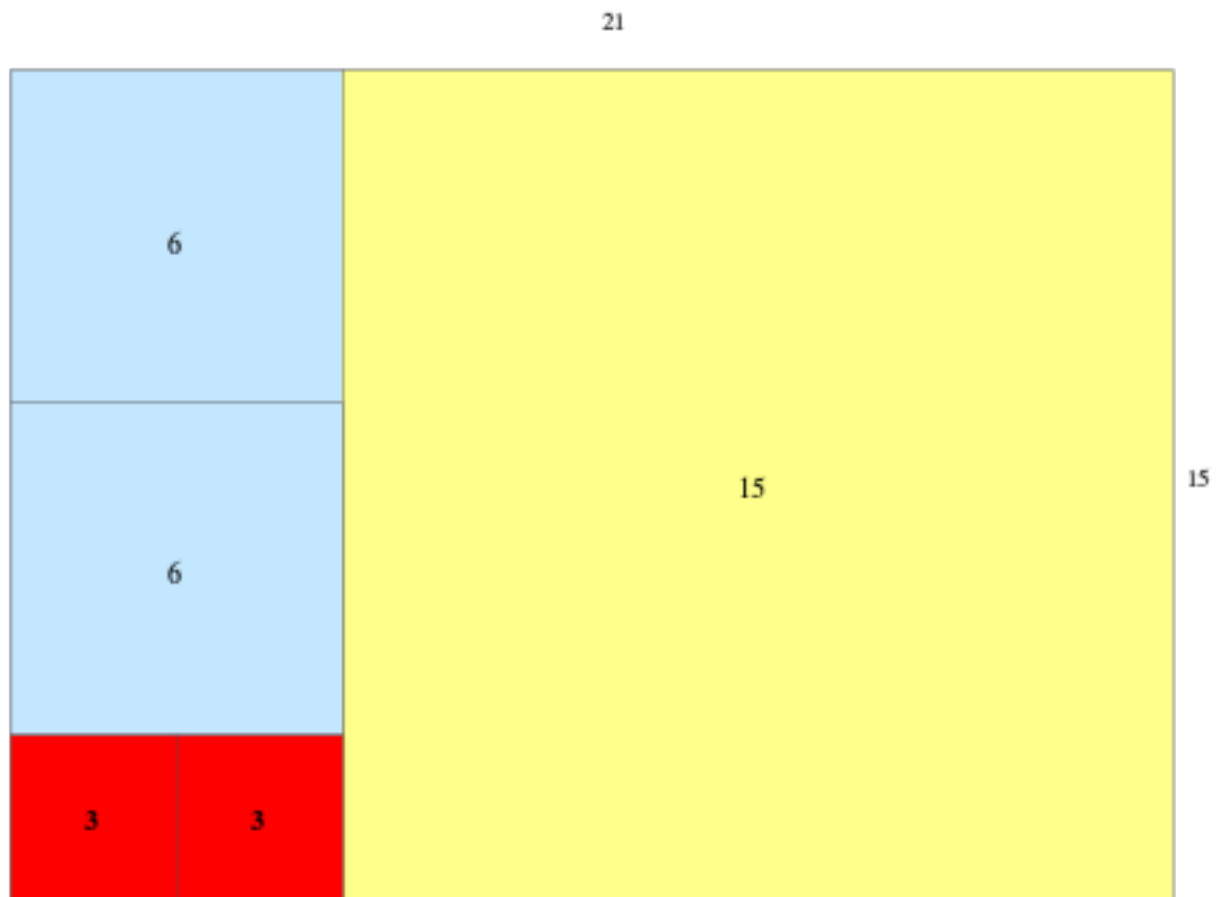
```
              // On pourrait convertir en base64 ou un array en json.
```

On écrit les fonctions :

```
/**
 * Creation of number e used to cipher
 * @param phii      (p-1) * (q-1)
 * @param pp        first prime number
 * @param qq        second prime number
 * @return the e number
 */
function primeOf(phii,pp,qq) {
    var min=Math.max(pp,qq)+1;
    for (var ee=min;ee<phii;ee++) {
        if (GCD(phii,ee)==1) break;
    }
    return ee;
}
```

et .../

```
/**
 * Compute the Greatest Common Denominator between a and b
 * @param a  First integer
 * @param b  Second integer
 * @return the GCD
 */
function GCD (a,b) {
    var r=a%b;
    while (r!=0) {
        a=b;
        b=r;
        r=a%b;
    }
    return b;
}
```

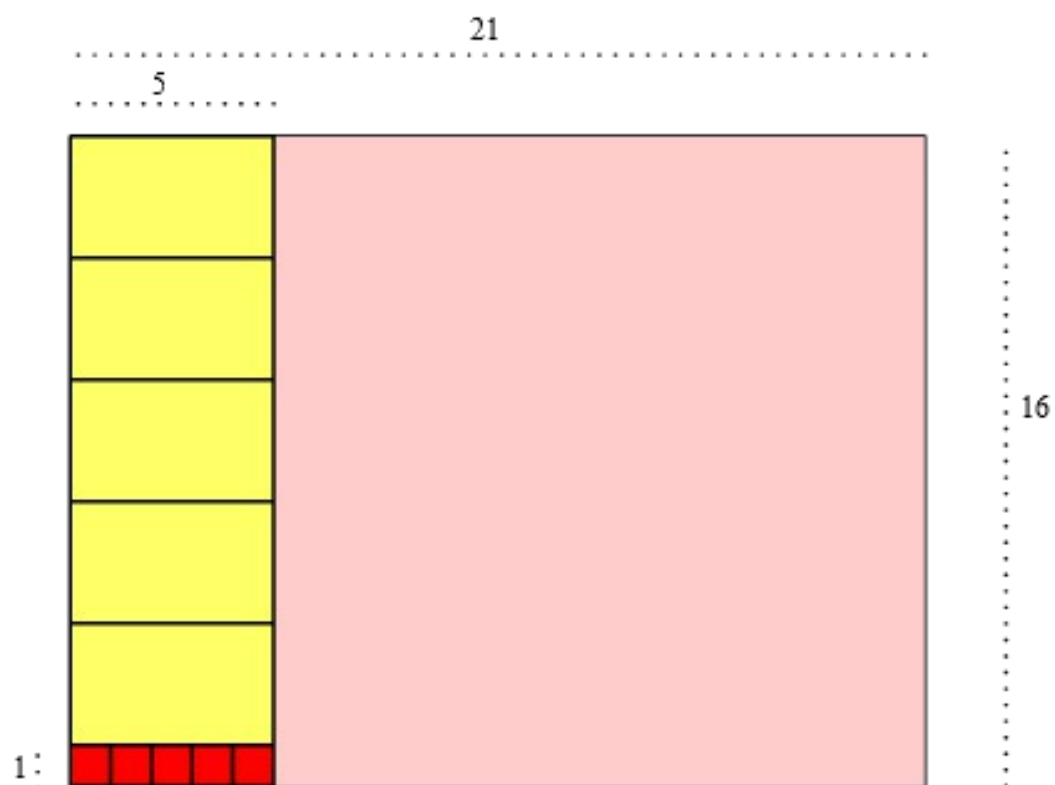



$$21 \% 15 = 6$$

$$15 \% 6 = 3$$

$$6 \% 3 = 0$$

$$\Rightarrow \text{PGCD} = 3$$



```

var r=a%b;
while (r!=0) {
    a=b;
    b=r;
    r=a%b;
}
return b;

```

```

a=21 ; b=16 ; r=5
?(r!=0)
    a=16
    b=5
    r=1
?(r!=0)
    a=5;
    b=1
    r=0
?(r!=0)

return 1

```

Théorie au tableau :

Principe de chiffrement :

Pour chaque caractères ou groupe de 2 caractères (16 bits) ,
on a la valeur dans la var

cp

on calcul la valeur chiffrée cc par

$$cc = (cp^e) \% n$$

ou e et n sont les composants de la clé publique.

Principe de déchiffrement :

On calcul à partir de ϕ , e , p , q et n ($p \cdot q$)

le 1er nombre d supérieur à p et q et inférieur à n (par une boucle)

qui vérifie :

$$(e \cdot d) \% \phi \text{ est égal à } 1$$

Ce qui donne la clé privé de déchiffrement :

(d,n)

Avec les valeurs

p =503 et q=563

n =283 189

ϕ =282 124

Clé privé

d=140313 et n=283 189

Je ne fais pas ici la démonstration mathématique

Voir

https://fr.wikipedia.org/wiki/Chiffrement_RSA

ou

<http://culturemath.ens.fr/maths/pdf/nombres/RSA.pdf>

On note juste qu'on ne peut pas retrouver cp avec e et cc car il y a une infinité de nombre qui donne le même reste de sa division par e.

Pratique :

Note :

Dans le prog. principal, on a :

```
// test si le bloc de 'by' octets est plus petit que n
if (Math.pow(256,by)>n) {
    var str="Public key n="+n+" too small !";
    log(str);
    throw(str);
}

// Bob a reçu (comme tout le monde) la clé publique,
// Prisonnier dans une tour par un dragon il envoie un message à Alice...
// il écrit son message à Alice
var plainTxt="Help me !";

// Il regroupe les octets de son message 2 par 2 dans un tableau.
var arrBy= toArrayOfBy(plainTxt,by);

// On a déjà vu cette fonction dans le programme avec clé secrète
// La seule différence est que les valeurs sont des BigNumber ()
arrOut.push(new BigNumber(c));
// car JS avec Math.pow(n,e) ; ne fonctionne pas avec des grands nombres.
// On utiliseras plus tard c.pow(n,e) ;

puis
// Chiffrement
var cipherArr=cipher (arrBy,e,n);
//
// puis
// Conversion du texte chiffrer en une string hexa
var cipheredTxt=toHexString(cipherArr,len16);
```

On va écrire la fonction *cipher* et étudier la fonction *toHexString* :

```
/**
 * Cipherng with public key (e,n)
 * @param arr      16 bits elements array with plain text
 * @param ee       Number e of public key
 * @param nn       Number n (p*q)
 * @return 16 bits elements array with ciphered text
 */

function cipher (arr,ee,nn) {
    BigNumber.config({ RANGE: 1000000 });
    BigNumber.config({ POW_PRECISION: 0 }) ; // No limit of significative digits
    var arrOut=[];
    for (var i in arr) {
        var cp=arr[i];
        // On applique
        var v=cp.pow(e);
        var cc=v.mod(nn) ;
        arrOut.push(cc);
    }
    return arrOut;
}
```

Cette fonction est déjà écrite. Explication.

```
/**
 * Creation of an hexa string before send ciphered text
 * @param arr          A text string
 * @param len16        Number hexa digits (len16=5)
 * @return Ciphered text as hexa string
 */
function toHexaString (arr,len16) {
    var hxChars="0123456789abcdef"; // Hexa digits from 0 to f
    var hxStr="";
    for (var i in arr) {                // for each value in array
        var v=arr[i]; var str16="";      // store in v
        while (v>0) {                    // while value hasn't been processed
            var d=v%16;                  // extract right value from 0 to 15
            var ch=hxChars.substr(d,1); // find corresponding hexa digit
            str16=ch+str16;              // store right to left the hexa digit
            v=Math.floor(v/16);          // put in v the left rest
        }
        for (var j=str16.length;j<len16;j++) {
            str16="0"+str16;             // zero left padding
        }
        hxStr+=str16;
    }
    return hxStr;
}
```

Note :

Alice reçoit le message (string hexa) de Bob.

Dans le prog. principal, on a :

```
// Alice calcul d donc sa clé privée (d,n) si ce n'est déjà fait.
var d=dCompute(phi,p,q,e);
```

```
// et
```

```
Convertit la string hexa en array
```

```
var decArr=toDecArray(cipheredTxt,len16);
```

```
// (qui au passage est égal à l'array chiffré de Bob)
```

.../

On va écrire la fonction *dCompute* et étudier la fonction *toDecArray* qui n'est que l'inverse de *toHexString*

```
/** Compute of number d of private key (d,n)
 * @param phii      (p-1) * (q-1)
 * @param pp        first prime number p
 * @param qq        second prime number q
 * @param e         Number e (idem in public key)
 * @return the d number
 */
function dCompute(phii,pp,qq,e) {
    var min=Math.max(pp,qq)+1;
    if (min==null || phii==null || phi<=min) {
        var str="error in  dCompute(phi,p,q) = (" +phii+", "+pp+", "+qq+)";
        log(str);
        throw(str);
    }
    for (var dd=min;dd<(p*q);dd++) {
        if ((e*dd)%phii==1) break;
    }
    return dd;
}
```

Cette fonction est déjà écrite. Explication.

```
/**
 * Convert hexa string ciphered text into array
 * @param str      hexa string
 * @param len16    Number hexa digits (len16=5)
 * @return Ciphered array
 */
function toDecArray (str,len16) {
    var arr=[];
    for (var i=0;i< str.length ; i+=len16) { // i target the first right digit of hexa string
        var str16=str.substr(i,len16);      // get the hexa string of 5 digits
        var v=Number('0x'+str16) ;          // convert to decimal
        arr.push(v);                         // store in returned array
    }
    return arr;
}
```

Note :

Alice déchiffre le message de Bob (array)

Dans le prog. principal, on a :

```
// déchiffrement
uncipherArr=uncipher(decArr,d,(p*q)) ;
// et
// Conversion de l'array déchiffré en text clair.
var unciphText=arrayByToString(uncipherArr,by);
```

On ne va écrire que la fonction *uncipher* car la fonction *arrayByToString* a déjà été vue dans exercice précédent.

```
/**
 * Unciphering with private key (d,n)
 * @param arr      16 bits elements array with ciphered text
 * @param dd       Number d of private key
 * @param nn       Number n (p*q)
 * @return 16 bits elements array with unciphered text
 */
function uncipher (arr,dd,nn) {
    BigNumber.config({ RANGE: 1000000000 }); // Number of significative digits
    BigNumber.config({ POW_PRECISION: 0 }); // No limit of significative digits
    //
    var arrOut=[];
    for (var i in arr) {
        var cc=new BigNumber(arr[i]);
        var v=cc.pow(d);
        var pc=v.mod(nn) ;
        //log("pc.toNumber()="+pc.toNumber());
        arrOut.push(pc.toNumber());
    }
    return arrOut;
}
```

Alice lit le message et décide sur son cheval blanc, de partir terrasser le dragon et délivré Bob !!

**** On teste ***