

# 1. Eléments de sécurité (HTTP , ...)

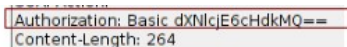
## 1.1. Basic Http Auth et limitations

### Positionnements possibles des informations d'authentification

#### En fin d'url :

`http://www.xy.com/zz/resource1?username=user1&password=pwd1`

Dans l'entête HTTP (avec **encodage base64** associé au standard "basic http auth") :



Authorization: Basic dXNlcjE6cHdkMQ==  
Content-Length: 264

← *décodage immédiat (quasi "en clair")*

### Limitations d'une authentification rudimentaire

Placer en fin d'url (ou bien dans l'entête HTTP) **une information d'authentification en clair (ou à peine cryptée via un encodage base64)** permet seulement de limiter l'accès aux utilisateurs qui connaissent le couple *username/password* .

Dans le cas où un "hacker" intercepte la requête et en récupère une copie, il connaît alors tout de suite le mot de passe et peut alors déclencher toutes les actions qu'il désire en se faisant passer par l'utilisateur "piraté" .

## 1.2. Cryptage élémentaire via hash (MD5 , SHA ) + salt

### "Basic Http Auth." ou fin d'URL avec hash(password)

Certains algorithmes standards de cryptage ("hachage") du mot de passe tels que "md5" ou "sha1" ou "..256" rendent très difficile le décryptage de celui-ci .

En véhiculant en fin d'URL (ou dans l'entête HTTP) le mot de passe haché

- celui-ci ne circule plus en clair (meilleure confidentialité).
- la base de données des mots de passe ne comporte que des informations cryptées et est donc moins vulnérable (si elle est piratée ou visualisée, les mots de passe "en clair" ne seront pas connus) .

#### Phases de la mise en oeuvre :

- Dès la saisie initiale du mot de passe, celui-ci est haché/crypté et stocké dans une base de données côté serveur.
- Lorsque le mot de passe est re-saisi côté navigateur , celui-ci est de nouveau haché/crypté (via le même algorithme) avant d'être véhiculé vers le serveur
- Le serveur compare les deux "hachages/cryptages" pour vérifier une authentification correcte

### "Basic Http Auth." avec "hash" et "salt"

- Un simple cryptage/hachage "md5" , "sha1" ou autre ne suffit souvent pas car il existe des bases de données d'associations entre mots de passe courants et les hachages correspondants "md5" ou "sha1" facilement accessibles depuis le web.
- D'autre part, si le mot de passe (en clair) peut ainsi être indirectement découvert, ceci peut être extrêmement problématique dans le cas où l'utilisateur utilise un même mot de passe pour plusieurs sites ou applications .
- De façon à prévenir les risques présentés ci-dessus, on utilise souvent un double encodage/cryptage prenant en compte une chaîne de caractère spécifique à l'application (ou à l'entreprise) appelée "salt" (comme grain de sel) .

Exemple :

cryptedPwd = md5OuSha1("my\_custom\_salt" + md5OuSha1(pwd)) ;

**"hash" (+ "salt") (récapitulatif)**

*AlgoCryptage* = **md5OuSha1(pwd)** ou bien  
**md5OuSha1("my\_custom\_salt" + md5OuSha1(pwd)) ;**

②a saisie du mot  
de passe en clair

②b cryptage via  
*algoCryptage*

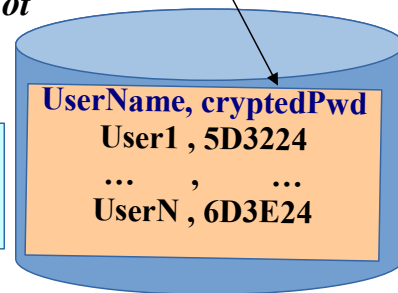
Coté client

③ transfert du mot de  
passe crypté via HTTP

④ comparaison du mot  
de passe crypté avec  
celui en base de  
données

Coté  
serveur

① stockage crypté  
en base de données  
dès la saisie initiale



Même s'il est indéchiffrable , si le mot de passe crypté  
est intercepté, l'authentification tombe à l'eau

### 1.3. Bcrypt pour crypter les mots de passe stockés en base

#### Algo. "bcrypt" pour les "password" stockés en base

Une authentification sérieuse consiste à utiliser conjointement HTTPS , des jetons et un algorithme de cryptage pour stocker les mots de passe en base.

L'algorithme "bcrypt" est tout à fait approprié pour crypter les mots de passe à stocker en base.

"Bcrypt" génère un "salt"/"clef" aléatoirement en fonction de n=10 ou autre et le résultat du cryptage n'est pas constant.

Bien que "non constant" et "avec clef jetée" , l'algorithme "bcrypt" peut déterminer si un ancien cryptage bcrypt récupéré en base correspond ou pas à un des cryptages possibles du mot de passe précisé en clair .

*// cryptage avant stockage :*

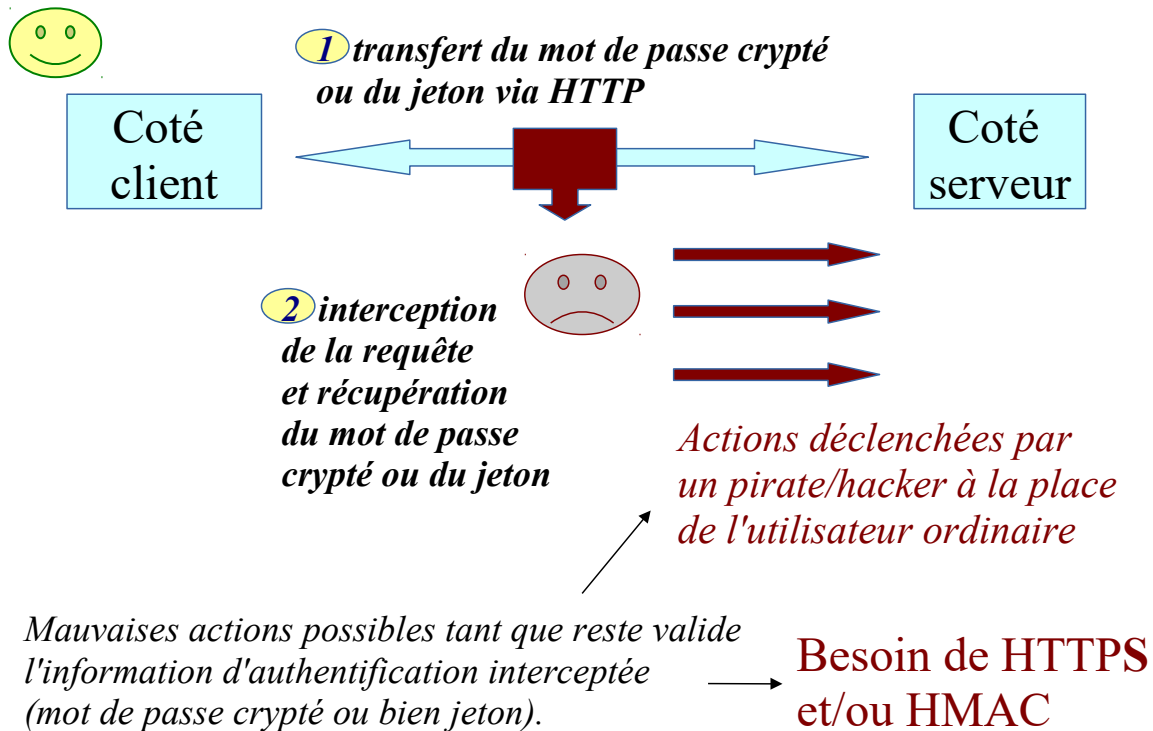
```
var bcryptedPwd =  
    bcrypt.hashSync(password, bcrypt.genSaltSync(12), null);
```

*// comparaison lors d'une authentification :*

```
if ( bcrypt.compareSync(password, bcryptedPwd) ) { ...} else { ...}
```

### 1.4. Problématique "man-in-the-middle"

## Problématique "man in the middle"



### 1.5. HMAC

## Signature des requêtes (avec clef secrète) et requête à usage unique (avec timestamp)

Pour éviter qu'une requête interceptée puisse conduire à une attaque de type "man in the middle" , on peut **ajouter une signature de requête** rendant celle-ci **inaltérable (non modifiable)** .

Dans le cas, où la requête serait interceptée , le "hacker" ne pourrait que la rejouer telle quelle (sans pouvoir la modifier).

**Pour**, tout de même **éviter, qu'une requête puisse être relancée plusieurs fois**, il suffit d'**ajouter un "timestamp" au message à envoyer**.

NB: Ces 2 techniques sont assez souvent utilisées ensembles et la technologie d'authentification associée s'appelle **HMAC** (keyed-hash message authentication code)

## HMAC avec timeStamp (partie 1 / coté "client")

1) l'application cliente prépare la requête (userName ou ... , timeStamp , paramètres , ...) . Celle-ci peut prendre la forme d'une URL en mode GET ou bien être la base d'un calcul d'empreinte en mode POST .

2) l'application cliente élabore une signature de requête :  
**signature=Base64(HMAC-SHA1(UTF-8-Encoding-Of(request), clef))**

Exemple (javascript) :

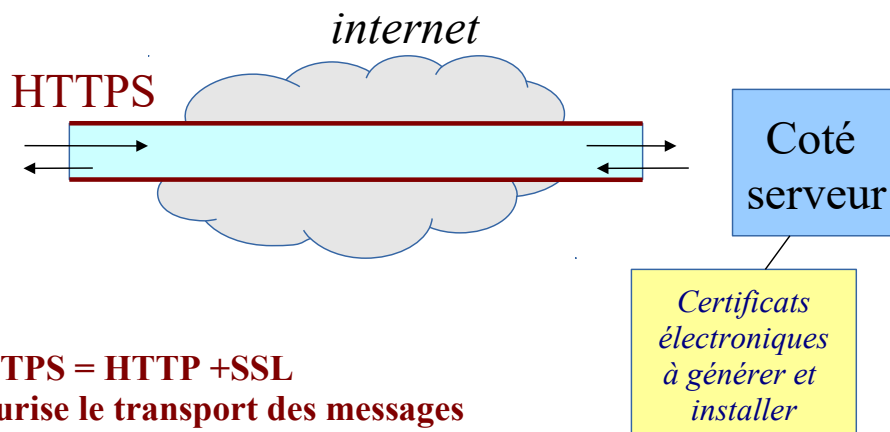
```
var user = "powerUser";    // Récupéré depuis la page d'authentification.
var password = "topSecret"; // Récupéré depuis la page d'authentification.
const salt = "13@!azerty"; // ou autre (selon application)
var encryptedPassword = CryptoJS.SHA1(CryptoJS.SHA1(password)+salt);
var httpVerb = "GET" ;
var currentTime = +new Date(); // valeur du timeStamp
var url = "http://www.xx.yy/product?user=" + user + "&timestamp=" + currentTime;
var httpUrl = httpVerb + ":" + url;
var signature =
    CryptoJS.HmacSHA1(httpUrl,encryptedPassword).toString(CryptoJS.enc.Base64);
url = url + "&signature=" + signature; //à envoyer
```

## **HMAC avec timeStamp (partie 2 / coté "serveur")**

- 1) l'application **serveur reçoit la requête** et en extrait le **username** (*en clair*).
- 2) l'application **serveur récupère en base le mot de passe crypté** (haché , salé) de l'utilisateur et s'en sert pour **calculer une signature du message avec le même algorithme** que du coté client .
- 3) l'application **serveur compare les deux signatures** (reçue et re-calculée) **pour vérifier que le message n'a pas été intercepté/modifié/altéré** .
- 4) l'application serveur tente de récupérer en base le dernier "timeStamp" associé à l'utilisateur s'il existe (véhiculé par requête précédente).  
**Si le timeStamp qui accompagne la requête n'est pas inférieur ou égal au dernier "timeStamp" récupéré en base tout va bien (la requête n'a pas été lancée plusieurs fois)** . Le timeStamp reçu est alors sauvegardé en base (pour le prochain test) , la requête est acceptée et traitée.

**Conclusion :** **HMAC avec timeStamp** garantit une **authentification robuste** mais ne gère pas la confidentialité des messages transmis, d'où l'éventuel besoin d'un complément HTTPS (HTTP + SSL) .

## **(HMAC avec timeStamp) ou ... + HTTPS**



**HTTPS = HTTP + SSL**  
**sécurise le transport des messages à travers le réseau internet**  
*(confidentialité , protection contre interception/modification, ...)* mais ne gère pas (tout seul) l'authentification.

