# Tutorial HTTP


**1 Introduction ..... 3**
1.1 Propos du document ..... 3
1.2 Introduction ..... 3
1.3 De HTTP 1.0 à HTTP 1.1 ..... 3

**2 URL HTTP ..... 4**
2.1 Format d'une URL HTTP ..... 4
2.2 Champs de l'URL HTTP ..... 4
2.3 Encodage d'URL ..... 4

**3 Méthodes HTTP ..... 5**

**4 Requête et réponse HTTP ..... 6**
4.1 Requête HTTP ..... 6
4.2 Réponse HTTP ..... 6

**5 En-têtes HTTP ..... 7**
5.1 En-têtes génériques ..... 7
5.2 En-têtes de la requête ..... 7
5.3 En-têtes de la réponse ..... 7
5.4 En-têtes *Hop-by-hop* et en-têtes *End-to-End* ..... 7
5.4.1 Hop-by-hop ..... 8
5.4.2 End-to-end ..... 8
5.5 Considérations sécuritaires concernants les en-têtes HTTP ..... 8

**6 Codes de statuts ..... 9**

**7 Connexions persistantes ..... 10**

**8 Cache ..... 11**
8.1 Deux types de cache ..... 11
8.1.1 Cache du browser ..... 11
8.1.2 Cache *proxy* ..... 11
8.2 En-têtes en relation avec le mécanisme de cache ..... 11
8.3 If-Modified-Since : Get conditionnel ..... 11
8.4 En-tête Cache-Control ..... 11
8.5 Utilisation de la méthode *Head* par un *proxy* ..... 12
*8.6 Etag* ..... 12

**9 Négociation de contenu ..... 13**
9.1 En-têtes concernant la négociation de contenu ..... 13
9.2 Exemple de négociation de contenu ..... 13
9.3 Interaction entre négociation de contenu et cache ..... 13
9.3.1 Hypothèse ..... 13
9.3.2 Solution ..... 14

**10 Cookies ..... 15**
10.1 Champs d'un *cookie* ..... 15
10.2 Sécurité ..... 16

**11 Authentification ..... 17**
*11.1 Basic* ..... 17
*11.2 Digest* ..... 17

**12 Redirection ..... 18**

**13 Analyse de protocole ..... 18**

**14 Encodage base 64 ..... 19**
14.1.1 Quand l'encodage base 64 est-il utilisé ? ..... 19
14.1.2 Principe ..... 19
14.1.3 Exemple : Conversion de « eig » ..... 19
14.1.4 Failles liées à l'encodage en base 64 : ..... 19

**15 Conclusion ..... 21**

**16 Sources ..... 21**

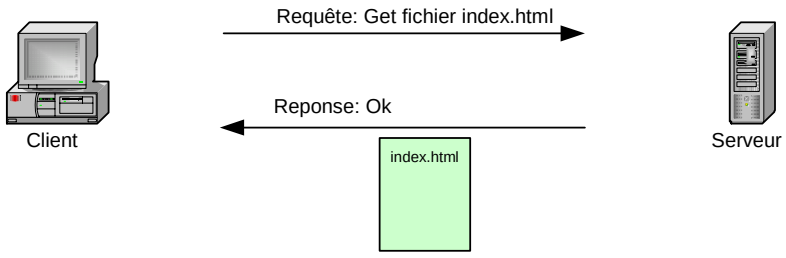
# 1 Introduction

## 1.1 Propos du document

Ce document a pour but de fournir les notions de base du protocole HTTP et de sensibiliser les lecteurs aux failles de sécurité inhérentes à ce protocole.
Ce document est basé en partie sur les travaux de diplôme de M. Tissot (EIVD 2003) et de M. Zanetta (EIG 2002).
Une acquisition *Ethereal* détaillée destinée à illustrer différents points de ce document se trouve au paragraphe § 13.

## 1.2 Introduction

HTTP (*Hypertext Transfer Protocol*) permet d'accéder aux fichiers situés sur le réseau Internet. Il est notamment utilisé pour le *World Wide Web*.
En matière de protocole, HTTP se place au dessus de TCP et fonctionne selon un principe de requête/réponse : le client transmet une requête comportant des informations sur le document demandé et le serveur renvoie le document si disponible ou, le cas échéant, un message d'erreur.




HTTP est un protocole sans connexion et chaque couple requête/réponse est de ce fait indépendant.

## 1.3 De HTTP 1.0 à HTTP 1.1

Ce protocole a été créé au CERN au début des années 1990 afin de fournir au *Web* un protocole de transfert simple.

Deux versions du protocole existent :

- HTTP 1.0 définit en 1996 par la RFC 1945
- HTTP 1.1 définit en 1999 par la RFC 2616

La version 1.1 apporte, entre autre, les améliorations suivantes :

- Cinq nouvelles méthodes (§ 3)
- Connexions persistantes (§ 7)
- *Digest Authentication* (§ 11.2)

## 2 URL HTTP

Un URI (*Uniform Resource Identifier*) est une chaîne de caractères structurée permettant d'**identifier** de manière unique une ressource dans un espace de nom donné.
Cette ressource peut être désignée soit par un URN (*Uniform Resource Name*) soit par un URL (*Uniform Resource Locator*). URN et URL sont des sous-ensembles de URI.
Un URN permet d'**identifier** une ressource par son nom même lorsque celle ci n'est plus disponible.
Un URL permet de **localiser** une ressource.
Dans le cas du protocole HTTP, une URL permet de localiser une page HTML, un fichier texte, un script cgi, une image...

### 2.1 Format d'une URL HTTP

Le format général d'une URL HTTP est le suivant :

```
HTTP://<host>:<port>/<path>?<query>#<fragment>
```

### 2.2 Champs de l'URL HTTP

| Champ | Description |
|---|---|
| host | Permet de spécifier le FQDN (ou adresse IP) du serveur possédant la ressource à accéder |
| port | Permet de spécifier le numéro de port à utiliser pour atteindre le serveur possédant la ressource. Si sa valeur est 80 (port par défaut du protocole HTTP), il n'est pas nécessaire de spécifier le numéro de port dans l'URL. |
| path | Permet de spécifier l'emplacement du fichier sur le serveur. Ce champ est en général constitué d'une suite de répertoires séparés par des '/' puis du nom du fichier à accéder. |
| query | Permet de passer un, ou plusieurs, paramètre(s) à un script PHP, Perl etc…. |
| fragment | Permet d'indiquer une « position » (ancre, *fragment*) dans une page |

### 2.3 Encodage d'URL

Les caractères ne pouvant être représentés dans une URL ( ';' '/' ' ?' '&' …) doivent être *escaped*.
Afin de rendre *escaped* un caractère (par exemple '&'), il faut remplacer ce caractère par son code ASCII codé en hexadécimal (26 pour '& ') et le faire précéder par le caractère '%'.
Le caractère '&' devient donc %26 et un espace ' ' devient donc %20.

Exemples :
http://www.exemple.com

http://www.exemple.com:80/News/Search?var1=janvier

http://www.exemple.com/Texte%20Avec%20Espace/Exemple.html

Pour plus d'informations concernant les règles d'encodage d'URL, se référer au document [5].

## 3 Méthodes HTTP

Les méthodes HTTP les plus fréquemment utilisées sont : GET, POST et HEAD
Cinq autres méthodes sont cependant définies par la version 1.1 du protocole.
Le tableau suivant détaille les méthodes HTTP disponible en fonction de la version du protocole.

| Méthodes | 1.0 | 1.1 | Description |
|---|---|---|---|
| Get | | | Permet de demander un document |
| Post | | | Permet de transmettre des données (d'un formulaire par exemple) à l'URL spécifiée dans la requête. L'URL désigne en général un script Perl, PHP… |
| Head | | | Permet de ne recevoir que les lignes d'en-tête de la réponse, sans le corps du document |
| Options | | | Permet au client de connaître les options du serveur utilisables pour obtenir une ressource |
| Put | | | Permet de transmettre au serveur un document à enregistrer à l'URL spécifée dans la requête |
| Delete | | | Permet d'effacer la ressource spécifiée |
| Trace | | | Permet de signaler au serveur qu'il doit renvoyer la requête telle qu'il la reçue |
| Connect | | | Permet de se connecter à un *proxy* ayant la possibilité d'effectuer du *tunneling* |

Certaines de ces méthodes nécessitent , en général, une authentification du client.

Pour plus d'informations concernant les méthodes HTTP, se référer aux paragraphes § 5.1.1 et § 9 du document [2].

# 4 Requête et réponse HTTP

## 4.1 Requête HTTP

La requête transmise par le client au serveur comprend :

1. une **ligne de requête** (*request-line*) contenant la méthode utilisée, l'URL du service demandé, la version utilisée de HTTP
2. une ou plusieurs **lignes d'en-têtes**, chacune comportant un nom et une valeur.

La requête peut optionnellement contenir une entité (contenu). Celle-ci est notamment utilisée pour transmettre des paramètres avec la méthode POST. L'entité est transmise après les lignes d'en-têtes, elle est séparée de la dernière en-tête par un double CRLF (*carriage return* et *linefeed*).

```
GET /index.html HTTP/1.1
Host: www.example.com
Accept: */*
Accept-Language: fr
User-Agent: Mozilla/4.0 (MSIE 6.0; windows NT 5.1)
Connection: Keep-Alive
```

Dans cet exemple, le client demande le document à l'adresse http://www.example.com/index.html, il accepte tous les types de document en retour, préfère les documents en français, utilise un navigateur (*browser*) compatible Mozilla 4.0 sur un système *Windows* NT 5.1 (*Windows XP*) et signale au serveur qu'il faut garder
la connexion TCP ouverte à l'issue de la requête (car il a d'autres requêtes à transmettre).

Pour plus d'informations concernant les requêtes HTTP, se référer au paragraphe § 5 de [2].

## 4.2 Réponse HTTP

La réponse transmise par le serveur au client comprend :

1. une **ligne de statut** (*status-line*) contenant la version de HTTP utilisée et un code d'état
2. une ou plusieurs **lignes d'en-têtes**, chacune comportant un nom et une valeur
3. Le **corps du document** retourné (les données HTML ou binaires par exemple). Une réponse ne contient pas obligatoirement un corps (exemple : sil s'agit d'une réponse à une requête HEAD, seule la ligne de statut et les en-têtes sont retournés.

La dernière ligne d'en-tête est suivie d'un double CRLF marquant le début du corps du document retourné (les données HTML ou binaires par exemple).

```
HTTP/1.1 200 OK
Date: Mon, 15 Dec 2003 23:48:34 GMT
Server: Apache/1.3.27 (Darwin) PHP/4.3.2 mod_perl/1.26
DAV/1.0.3
Cache-Control: max-age=60
Expires: Mon, 15 Dec 2003 23:49:34 GMT
Last-Modified: Fri, 04 May 2001 00:00:38 GMT
ETag: "26206-5b0-3af1f126"
Accept-Ranges: bytes
Content-Length: 1456
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html>
...
```

Dans cet exemple, le code 200 nous indique que le document demandé a été trouvé.
Pour faciliter la gestion du cache du client, le serveur transmet la date actuelle, la date de dernière modification du document et la date d'expiration (après laquelle le document doit être demandé à nouveau).
L'en-tête *Content-Type* nous apprend que le document retourné est de type HTML et l'en-tête *Content-Length* indique que le corps du document a une longueur de 1456 octets.
L'en-tête *Server* renseigne sur le logiciel serveur utilisé. L'envoi d'une telle information n'est pas recommandé d'un point de vue sécuritaire.

Pour plus d'informations concernant les requêtes HTTP, se référer au paragraphe § 6 de [2].

# 5 En-têtes HTTP

## 5.1 En-têtes génériques

Certains en-têtes peuvent se trouver aussi bien dans la requête que dans la réponse. Les principaux sont donnés dans le tableau ci-dessous :

| Champ | Description |
|---|---|
| Content-length | Longueur en octets des données suivant les en-têtes |
| Content-type | Type MIME des données qui suivent |
| Connection | Indique si la connexion TCP doit rester ouverte (*Keep-Alive*) ou être fermée (*close*) |

## 5.2 En-têtes de la requête

Les en-têtes suivants n'existent que dans les requêtes HTTP. Seul l'en-tête *Host* est cependant obligatoire dans la version 1.1 de HTTP.
La version 1.0 de HTTP ne posséde pas d'en-têtes obligatoires.

| Champ | Description |
|---|---|
| Accept | Types MIME que le client accepte |
| Accept-encoding | Méthodes de compression supportées par le client |
| Accept-language | Langues préférées par le client (pondérées) |
| Cookie | Données de *cookie* mémorisées par le client |
| Host | Hôte virtuel demandé |
| If-modified-since | Ne retourne le document que si modifié depuis la date indiquée |
| If-none-match | Ne retourne le document que sil a changé |
| Referer | URL de la page à partir de laquelle le document est demandé |
| User-agent | Nom et version du logiciel client |

Pour plus d'informations concernant les en-têtes des requêtes HTTP, se référer au paragraphe § 5.3 de [2].

## 5.3 En-têtes de la réponse

| Champ | Description |
|---|---|
| Allowed | Méthodes HTTP autorisées pour cette URI (comme POST) |
| Content-encoding | Méthode de compression des données qui suivent |
| Content-language | Langue dans laquelle le document retourné est écrit |
| Date | Date et heure UTC courante |
| Expires | Date à laquelle le document expire |
| Last-modified | Date de dernière modification du document |
| Location | Adresse du document lors d'une redirection |
| Etag | Numéro de version du document |
| Pragma | Données annexes pour le navigateur (par exemple, no.cache) |
| Server | Nom et version du logiciel serveur |
| Set-cookie | Permet au serveur d'écrire un *cookie* sur le disque du client |

Pour plus d'informations concernant les en-têtes des réponses HTTP, se référer au paragraphe § 6.2 de la RFC 2616 [2].

## 5.4 En-têtes *Hop-by-hop* et en-têtes *End-to-End*

Les en-têtes détaillés dans le § 5 peuvent être de deux types : *Hop-by-hop* ou *End-to-end*.

### 5.4.1 Hop-by-hop

Ces en-têtes sont destinés à être examinés par tout nœud recevant le paquet.

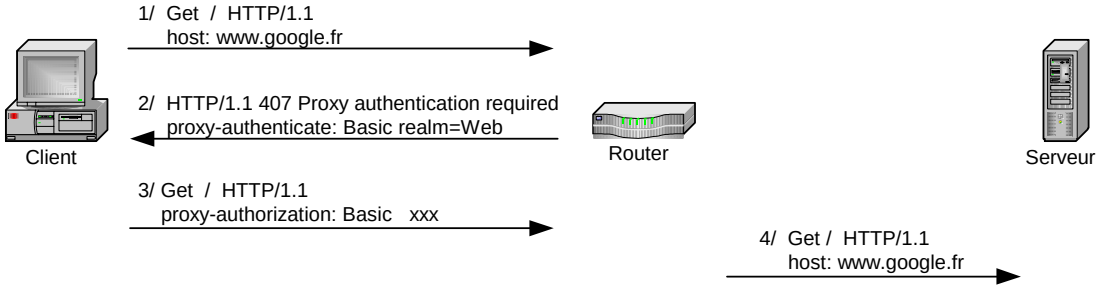
Les en-têtes suivants sont de type *Hop-by-hop* :

- *Connection*
- *Proxy-authenticate*
- *Proxy-authorization*
- *TE*
- *Trailers*
- *Transfer-Encoding*
- *Upgrade*

#### 5.4.1.1 Exemple

Prenons le cas d'un client devant passer par un *proxy* afin de pouvoir envoyer des requêtes HTTP sur Internet.
Le *proxy* nécessite une authentification du client et va pour se faire utiliser l'en-tête *Proxy-authenticate*.

1. => Le client va donc établir une connexion TCP entre lui et le *proxy* puis il va envoyer au *proxy* sa requête HTTP.
2. => Le client n'étant pas encore authentifié, le *proxy* va lui retourner une réponse possédant l'en-tête *proxy-authenticate*.
3. => Le client réitère sa requête en y ajoutant les données d'authentification demandées par le serveur. Les données d'authentification sont placées dans l'en-tête *proxy-authorization*.
4. => Une fois le client correctement authentifié, le *proxy* établit alors une connexion TCP avec le serveur demandé puis lui transmet la requête émise par le client. L'en-tête *proxy-authorization* n'est pas transmise par le *proxy*.




Il apparaît clairement que les en-têtes *proxy-authenticate* et *proxy-authorization* n'ont de sens que pour la communication entre le client et le *proxy*.
Ces en-têtes ne doivent pas être transmis.

### 5.4.2 End-to-end

Ces en-têtes sont destinés au destinataire final de la requête (ou de la réponse) HTTP. De ce fait, ces en-têtes doivent être ignorés par les nœuds (*proxies*, routeurs…) constituant le chemin suivi par le paquet. Les nœuds vont donc simplement transmettre ces en-têtes au destinataire final.

Tous les en-têtes sont de type *end-to-end* **exceptés** ceux présents dans le paragraphe § 5.4.1.

## 5.5 Considérations sécuritaires concernants les en-têtes HTTP

Les en-têtes permettent de fournir aux deux protagonistes d'un échange HTTP des informations afin d'optimiser (cache, connection *keep-alive…*) et de personnaliser (négociation de contenu, *cookies*…) le contenu échangé.
Cependant, certaines informations transmises dans ces en-têtes ne devraient pas être divulguée pour des raisons de sécurité.
C'est le cas de l'en-tête *Server*.
En effet, la première action entreprise par un *hacker* lors d'une tentative d'intrusion d'un système consiste à identifier le système en question.
Cette identification peut être effectuée par prise d'empreinte du système grâce à un outil du type NMAP (ou P0f pour une prise d'empreinte passive) ou, plus simplement, en regardant la valeur de l'en-tête *Server* reçue après avoir effectué une requête HTTP.
Par défaut, un serveur Web IIS 6 aura pour en-tête *Server* :

```
Server: Microsoft-IIS/6.0
```

Solution :
L'en-tête *Server* (tout comme d'autres en-têtes) peut être modifié à l'aide d'un *proxy* applicatif (*Blue Coat* par exemple, cf. [8] § 4.3.4) afin d'afficher une information fausse ou de laisser l'en-tête *Server* vide.

## 6 Codes de statut

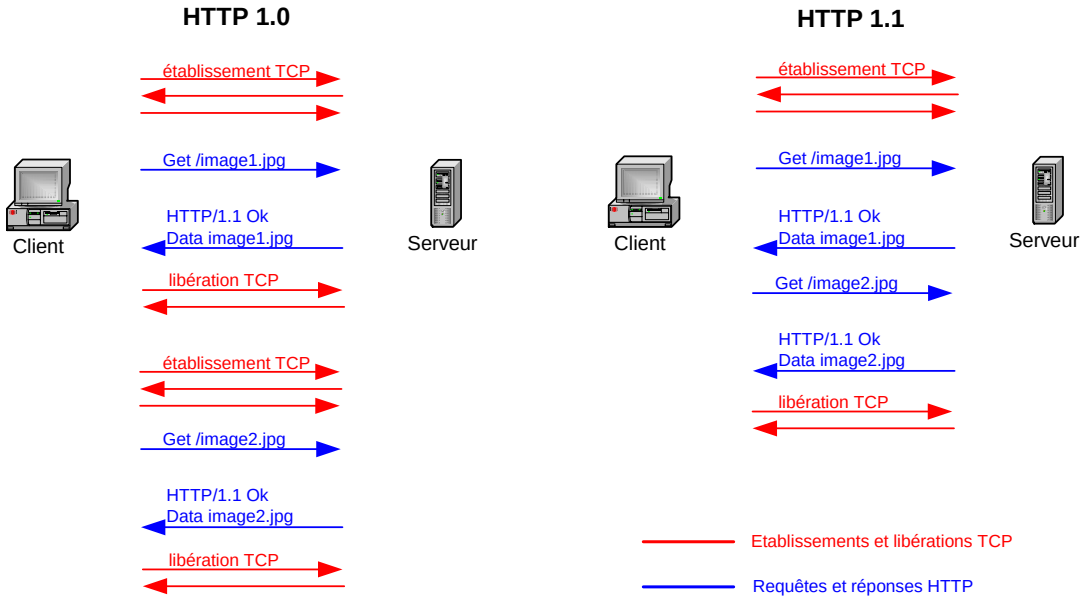
Lorsque le serveur renvoie un document, il lui associe un code de statut renseignant ainsi le client sur le résultat de la requête (requête invalide, document non trouvé…).
Les principales valeurs des codes de statut HTTP sont détaillées dans le tableau ci-après.

| Code | Nom | Description |
|---|---|---|
| **Information 1xx** | | |
| 100 | Continue | Utiliser dans le cas où la requête possède un corps. |
| 101 | Switching protocol | Réponse à une requête |
| **Succès 2xx** | | |
| 200 | OK | Le document a été trouvé et son contenu suit |
| 201 | Created | Le document a été créé en réponse à un PUT |
| 202 | Accepted | Requête acceptée, mais traitement non terminé |
| 204 | No response | Le serveur n'a aucune information à renvoyer |
| 206 | Partial content | Une partie du document suit |
| **Redirection 3xx** | | |
| 301 | Moved | Le document a changé d'adresse de façon permanente |
| 302 | Found | Le document a changé d'adresse temporairement |
| 304 | Not modified | Le document demandé n'a pas été modifié |
| **Erreurs du client 4xx** | | |
| 400 | Bad request | La syntaxe de la requête est incorrecte |
| 401 | Unauthorized | Le client n'a pas les privilèges d'accès au document |
| 403 | Forbidden | L'accès au document est interdit |
| 404 | Not found | Le document demandé n'a pu être trouvé |
| 405 | Method not allowed | La méthode de la requête n'est pas autorisée |
| **Erreurs du serveur 5xx** | | |
| 500 | Internal error | Une erreur inattendue est survenue au niveau du serveur |
| 501 | Not implemented | La méthode utilisée n'est pas implémentée |
| 502 | Bad gateway | Erreur de serveur distant lors d'une requête *proxy* |

Pour plus d'informations concernant les codes de statut, se référer aux paragraphes § 6.1.1 et § 10 de la RFC 2616.

## 7 Connexions persistantes

La fonctionnalité de connexions persistantes apparue avec la version 1.1 du protocole HTTP est sans doute l'amélioration la plus notable par rapport à la version 1.0.
Avec la version 1.0 de HTTP, une nouvelle connexion TCP doit être établie pour chaque URL demandée.
Avec la version 1.1, une connexion TCP existante peut être réutilisée pour demander d'autres URLs.
De plus, il est possible avec la version 1.1 d'HTTP d'effectuer sur une seule connexion TCP plusieurs requêtes HTTP sans avoir à attendre les réponses HTTP (*Pipelining*).




Cette amélioration permet de :

- diminuer la charge CPU (pour les routeurs, serveurs *Web*, clients, *proxies*…) engendrée par les établissements TCP
- diminuer le nombre de paquets transitant sur le réseau dû aux établissements TCP
- diminuer la latence engendrée par l'établissement de connexion TCP

Les clients désirant maintenir une connexion TCP ouverte doivent inclure dans leur requête HTTP l'en-tête *Connection*: *keep-alive*

Pour plus d'informations concernant les connexions persistantes, se référer au paragraphe § 8.1 de la RFC 2616.

# 8 Cache

Une page HTML est souvent constituée d'informations (textes ou images) ne subissant aucune modification pendant plusieurs jours.
Il devient alors intéressant de mettre en cache les objets pour lesquels nous avons déjà effectué une requête.

## 8.1 Deux types de cache

### 8.1.1 Cache du navigateur

Les *browsers* actuels possèdent, normalement, tous la fonctionnalité de cache. Ce cache va permettre au *browser* de fournir immédiatement les objets présents dans le cache sans avoir à effectuer une requête auprès du serveur possédant l'objet..
Dans le cas d'*Internet Explorer 6*, les réglages conernant le cache de ce navigateur sont disponibles de la manière suivante :

- `Start > Settings > Control panel > Internet options`
- Dans l'onglet `General`, cliquer sur `Settings`

### 8.1.2 Cache *proxy*

Les *proxies* HTTP possèdent souvent, entre autre, la fonctionnalité de cache. Ce cache va permettre au *proxy* de fournir aux clients les objets présents dans son cache sans qu'il doive effectuer une requête HTTP auprès du serveur possédant l'objet.
Ce cache est à la disposition de tous les clients utilisant le *proxy*.

## 8.2 En-têtes en relation avec le mécanisme de cache

Réponse HTTP :

```
HTTP/1.1 200 Ok
Content-Type: image/gif
Server: GWS/2.1
Content-length: 8866
Last-Modified: Tue, 23 Sep 2003 03:21:11 GMT
Expires: Sun, 17 Jan 2038 19:14:07 GMT
Date: Wed, 14 Apr 2004 13:20:30 GMT
Data..
```

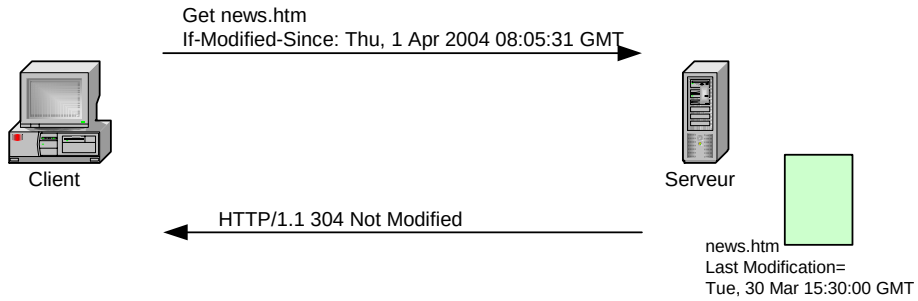
Comme nous pouvons le constater, la réponse HTTP contenant l'objet demandé est accompagnée de plusieurs en-têtes dont certains permettent d'agir sur le fonctionnement du cache.
Ces en-têtes sont (liste non exhaustive) :

- *Last-Modified*
- *If-Modified-Since*
- *Date*
- *Expires*
- *Cache-Control*
- *ETag*

Les 4 premiers en-têtes étaient déjà présents dans la version 1.0 de HTTP.
Les en-têtes *Cache-Control* et *ETag* sont des nouveautés de la version 1.1 du protocole.

## 8.3 If-Modified-Since : Get conditionnel

Un *Get* conditionnel peut être effectué en incluant l'en-tête *If-Modified-Since* dans une requête *Get* HTTP.
Cet en-tête permet de spécifier au serveur HTTP que l'on ne désire recevoir le fichier demandé que sil a subit des modifications depuis la date indiquée dans l'en-tête.




## 8.4 En-tête Cache-Control

Cet en-tête permet un contrôle accru des mécanismes de cache et offre une gestion plus efficace du cache.
Plusieurs directives peuvent être placées dans un en-tête *Cache-Control*.

Exemples de directives :

| Directive | Description |
|---|---|
| *max-age* | Spécifie le temps (en secondes) maximum durant lequel l'objet peut être considéré comme « frais » |
| *private* | Indique que la réponse ne doit pas être mise en cache |
| *public* | Indique que la réponse peut être mise en cache par n'importe quel cache |

La liste exhaustive des directives existantes pour l'en-tête *Cache-Control* est disponible au paragraphe § 14.9 du document [2].

### *8.5* Utilisation de la méthode *Head* par un *proxy*

Comme vu au paragraphe § 3 de ce document, la méthode *Head* permet de ne récupérer que les lignes d'en-têtes d'une réponse à un *Get*.
Cette méthode permet des gains de performance non négligeable dans certains cas.

Exemple :
Un *proxy* utilisé par les employés de la société X possède la fonction de cache HTTP. Les fichiers les plus fréquemment consultés sont donc présents dans son cache et peuvent être retournés directement aux employés lorsqu'ils sont demandés.
Afin de ne pas fournir des documents « périmés », le *proxy* doit régulièrement vérifier si une nouvelle version du fichier stocké en cache n'est pas disponible.
Pour ce faire, le *proxy* peut effectuer une requête *Head* sur le fichier en question afin d'obtenir la date de dernière modification du fichier (en-tête *Last-Modified*). Si cette date est plus récente que la date du fichier présent en cache, alors le *proxy* va effectuer une requête *Get* sur le fichier afin de posséder la dernière version du fichier. Sinon la version du fichier présente en cache est considérée comme « fraîche » et donc conservée.
L'utilisation de la méthode *Head* permet ainsi d'éviter la transmission du fichier lorsque celle ci est inutile.

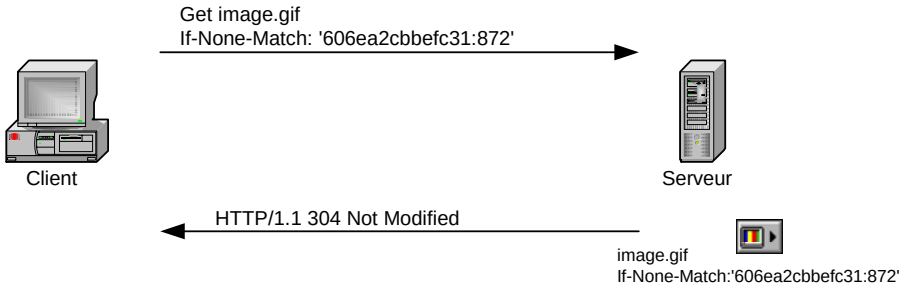
### *8.6* *Etag*

Un *ETag* est une valeur générée par le serveur Web permettant d'identifier de manière unique un objet.

Exemple : ETag: ‘ 606ea2cbbefc31:872 ‘

Un *ETag* est créé lorsqu'un nouvel objet est mis à disposition sur le serveur Web.
Cet *ETag* est modifié à chaque modification de l'objet en question.

Un *Etag* est souvent utilisé avec les en-têtes *If-Match* et *If-None-Match*.




Pour plus d'informations concernant le mécanisme de cache HTTP, se référer au document [5].

# 9 Négociation de contenu

La négociation de contenu a pour but de fournir à un client la représentation la mieux adaptée à ses besoins.

Exemple :
Un site portail international propose des informations en plusieurs langues. L'utilisateur arrive sur le site sur une page `index.html` où il doit sélectionner la langue dans laquelle il veut consulter les informations.
Il serait intéressant de pouvoir automatiser cette phase et donc, de fournir à l'utilisateur la représentation appropriée des informations, sans qu'il ne doive sélectionner la langue désirée.

La version 1.1 de HTTP permet cette automatisation grâce à l'en-tête *Accept-Language*.
Cet en-tête est, normalement, présent dans toute requête HTTP.

Cet en-tête accepte plusieurs valeurs simultanément.
Exemple : `Accept-Language: fr, en`

Il est également possible de pondérer ses préférences.
Exemple : `Accept-Language: fr ; q=1.0 , en ; q=0.8`

Ces principes sont applicables aux jeux de caractères, types de fichiers textuel (HTML ou .txt), types d'images (JPEG ou GIF)…

Plus d'informations concernant la négociation de contenu est disponible au paragraphe § 12 de [2].

Des informations concernant la négociation de contenu et l'algorithme de négociation de contenu utilisé par *Apache* sont disponibles dans le document [6].

## 9.1 En-têtes concernant la négociation de contenu

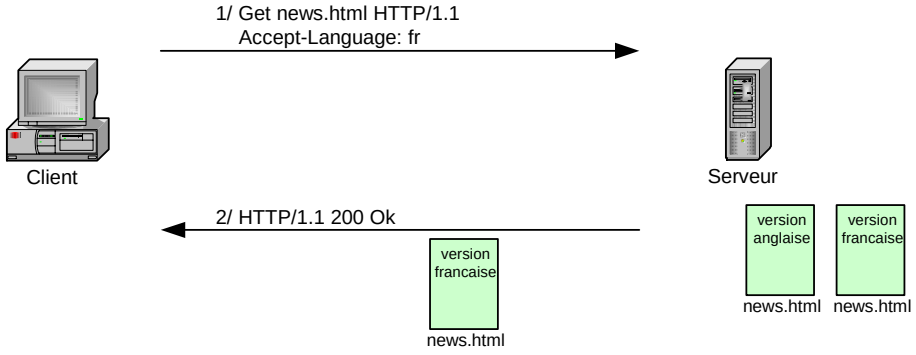
| En-tête | Description |
|---|---|
| *Accept* | Permet de spécifier les préférences concernant les types audio, textuels, images… |
| *Accept-Language* | Permet de spécifier les préférences concernant les langues |
| *Accept-Charset* | Permet de spécifier les préférences concernant les jeux de caractères |
| *Accept-Encoding* | Permet de spécifier les préférences concernant les compressions et encodages |

Pour plus d'informations concernant ces en-têtes, voir les paragraphes § 14.1, § 14.2, § 14.3 et § 14.4 du document [2].

## 9.2 Exemple de négociation de contenu

1. Le client effectue une requête afin d'obetnir le fichier `news.html`. Il spécifie également que le fichier voulu doit lui être retourné en français en priorité.
2. Le serveur possède deux versions du fichier : - version française
   - version anglaise

   Ayant connaissance des préférences du client grâce à l'en-tête *Accept-Language* reçu, le serveur va retourner la version française du document.




## 9.3 Interaction entre négociation de contenu et cache

### 9.3.1 Hypothèse

Les employés de la société X passent par un *proxy* HTTP, pourvu de la fonctionnalité de cache, pour *surfer* sur le Web.
Un utilisateur A effectue une requête pour obtenir le fichier `news.html`, en version anglaise, du serveur Y.

Ce fichier est placé en cache par le *proxy*.
Un utilisateur B effectue une requête pour obtenir le même fichier `news.html`, en version française, du serveur Y.
Le *proxy* va retourner immédiatement le fichier en cache sans envoyer de nouvel requête au serveur Y.
L'utilisateur B ne pourra donc pas obtenir le fichier dans la langue désirée.

### 9.3.2 Solution

Il faut donc utiliser, par exemple, l'en-tête `Cache-Control: private` afin d'empêcher la mise en cache des documents obtenus à l'aide d'un mécanisme de négociation de cache.
Par défaut, un serveur *Apache* empêche la mise en cache de tels documents.

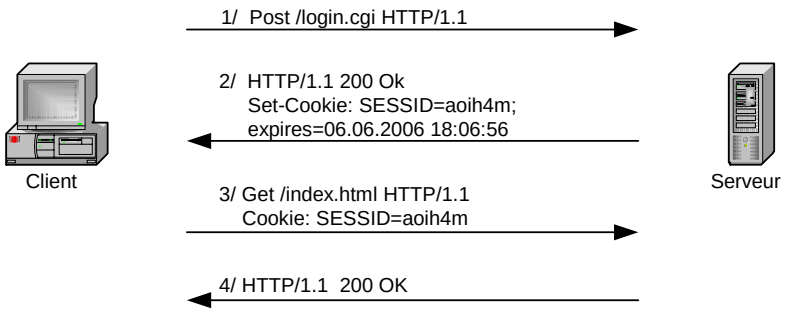
# 10 Cookies

Comme précisé dans le paragraphe § 1.2, le protocole HTTP ne possède pas de mécanisme de session. Pour pallier ce manque, *Netscape* a créé les *cookies* afin de fournir à HTTP un mécanisme de gestion d'états.
Les *cookies* vont permettre au serveur de mémoriser des données du côté client. Cela permet un suivi de l'utilisateur d'une requête à l'autre et d'implémenter ainsi la notion de session.
Afin de donner l'ordre au client de créer un *cookie*, le serveur place un en-tête *Set-Cookie* dans sa réponse comprenant :

- le nom du *cookie*
- sa valeur
- son contexte (*path*)
- sa date d'expiration

Le schéma de séquence suivant symbolise un client qui, après avoir rempli un formulaire (*username/password*) :

1. Envoi (*Post*) les données à un script login.cgi
2. Le serveur informe le client que les données ont été correctement reçues par le script (200 Ok) et demande au client de créer un cookie ayant pour valeur SESSID=aoih4m et expirant le 06.06.2003 à 18:06:56.
3. Le client effectue une requête pour obtenir le fichier `index.html` en joignant à la requête le *cookie* créé en 2.
4. Après avoir vérifié que la valeur SESSID correspondait bien à une session valide, le serveur renvoie le fichier demandé au client




Un *cookie* est souvent utilisé afin de mémoriser :

- une clé de session à validité limitée
- les préférences de l'utilisateur concernant un site *Web*
- les achats réalisés par l'utilisateur etc…

## 10.1 Champs d'un *cookie*

`'NAME'` : Obligatoire, `NAME` est le nom attribué par le serveur au *cookie*.

`'Value'` : Obligatoire, `VALUE` est une valeur attribuée au *cookie* par le serveur et est théoriquement dénuée de sens pour l'utilisateur. Cette valeur sert à identifier l'utilisateur au serveur.

`Comment='value'`: Optionnel, comme les *cookies* peuvent être utilisés pour enregistrer des informations sensibles sur un utilisateur, ce champ est utilisé par le serveur pour décrire comment il va utiliser ce *cookie*. L'utilisateur peut inspecter ce champ pour décider s'il veut continuer les échanges avec le serveur ou pas.

`CommentURL='URL'` : Optionnel, idem que précédemment sauf que dans ce cas, les informations concernant l'utilisation du *cookie* faite par le serveur sont disponibles à l'url 'URL'

`Discard` : Optionnel, sert à informer le navigateur qu'il peut ignorer le *cookie* dès que l'utilisateur termine sa session.

`Domain= 'value'` : Décrit comme optionnel par la RFC 2965 mais en réalité obligatoire, sert à spécifier le domaine pour lequel le *cookie* est valide. Il est utilisé par le navigateur pour savoir à qui (serveur) le *cookie* doit être envoyé.

`Max-Age= 'value'` : Optionnel, donne l'age maximum du *cookie* avant qu'il ne soit "périmé". Sa valeur est en seconde, depuis le 1er janvier 1970 00:00:00 GMT. Lorsque ce champ n'est pas spécifié, le navigateur mémorise le *cookie* et le détruit lorsque l'utilisateur ferme le navigateur. Un tel *cookie* est dit ***de session*** alors qu'un *cookie* possédant une valeur définie de *Max-Age* est dit ***persistant***.

`Path= 'value'` : Optionnel, spécifie le chemin sur le serveur auquel le *cookie* s'applique.

`Port= 'portlist'` : Optionnel, sert à restreindre un *cookie* à un ou plusieurs port(s) spécifique(s).

`Secure` : Optionnel, c'est un booléen qui spécifie si une connexion SSL est nécessaire pour accéder au *cookie*.

`Version= 'value'` : Obligatoire, ce champ identifie la version du mécanisme de gestion d'état.

Pour plus d'informations concernant les *cookies*, voir le document [4]

Sur un système XP, il est possible de visualiser les *cookies* à partir de C:\Documents and Settings*user*\Cookies et de les supprimer (Start > Control Panel > Internet Options > Delete Cookies...).

### 10.2 Sécurité

Les *cookies* ne présentent pas un danger en eux mêmes. Ils peuvent cependant générer des failles dans le cas où, par exemple, ils sont utilisés pour authentifier un utilisateur X en contenant le numéro de session.
Dans un tel cas, une personne Y mal intentionnée pourrait voler le *cookie* et se connecter au site en étant authentifié en tant que X.
Les *cookies* sont également souvent utilisés dans les attaques de type *Cross-Site Scripting* (cf. http://www.cgisecurity.com/articles/xss-faq.shtml pour plus d'informations sur le *Cross-Site Scripting*).

# 11 Authentification

L'accès à certains documents peut nécessiter une authentification du client.
Dans la version 1.0 du protocole HTTP, seule l'authentification de type *Basic* était disponible.
La version 1.1 ajoute l'authentification de type *Digest*.

## *11.1 Basic*

1. => GET http://www.google.ch HTTP/1.1

Pour consulter une page Web, le navigateur envoie une requête HTTP au serveur contenant l'URL du document à obtenir.

2. <= HTTP/1.1 401 *Unauthorized*
   WWW-Authenticate: BASIC realm="private"

Le serveur va retourner au navigateur une réponse HTTP possédant un code de statut 401 (*Unauthorized*) avec l'en-tête `www-authenticate`. Cet en-tête est suivie par le nom de la méthode d'authentification à utiliser (*Basic*) et le royaume (*realm*) auprès du quel l'authentification doit être faite. Le schéma d'autorisation est généralement le schéma BASIC.
BASIC = Encodage_Base_64(*username*:*password*) (cf. § 14 pour plus d'informations concernant l'encodage au format base 64)
L'en-tête `www-authenticate` indique également auprès de quel domaine (*realm*) l'utilisateur doit s'authentifier.

3. => GET http://www.google.ch HTTP/1.1
   Authorization: BASIC xxx

Le client réitère sa requête en y ajoutant son *username*:*password* (séparés par ' : ') encodé au format BASIC, comme demandé par l'en-tête *WWW*-Authenticate.
BASIC = Encodage_Base_64(*username*:*password*)

4. <= HTTP/1.1 200 OK
   Data…..

Une fois le client correctement authentifié par le serveur, ce dernier enverra au client les ressources demandées.

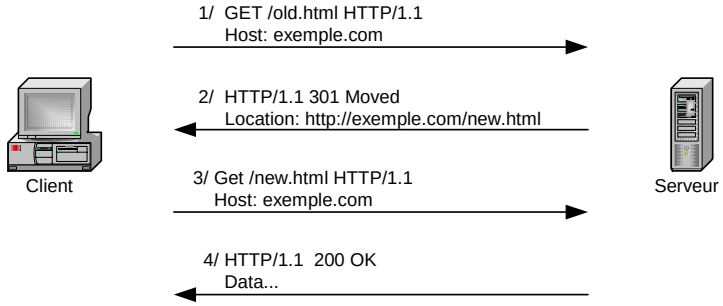

## *11.2 Digest*

L'authentification de type *Digest* est basée sur l'authentification de type *Basic* et sur un mécanisme de *Challenge/Response*.
Le *Challenge/Response* permet d'éviter la transmission du mot de passe sur le réseau (c'est un *hash* MD5 du mot de passe et du *challenge* qui est transmis à la place) ainsi que les *replay attacks* (grâce à la valeur aléatoire qu'est le *challenge*).

Pour plus d'informations concernant l'authentification HTTP de type Digest, se référer au paragraphe § 3 du document [3].

## 12 Redirection

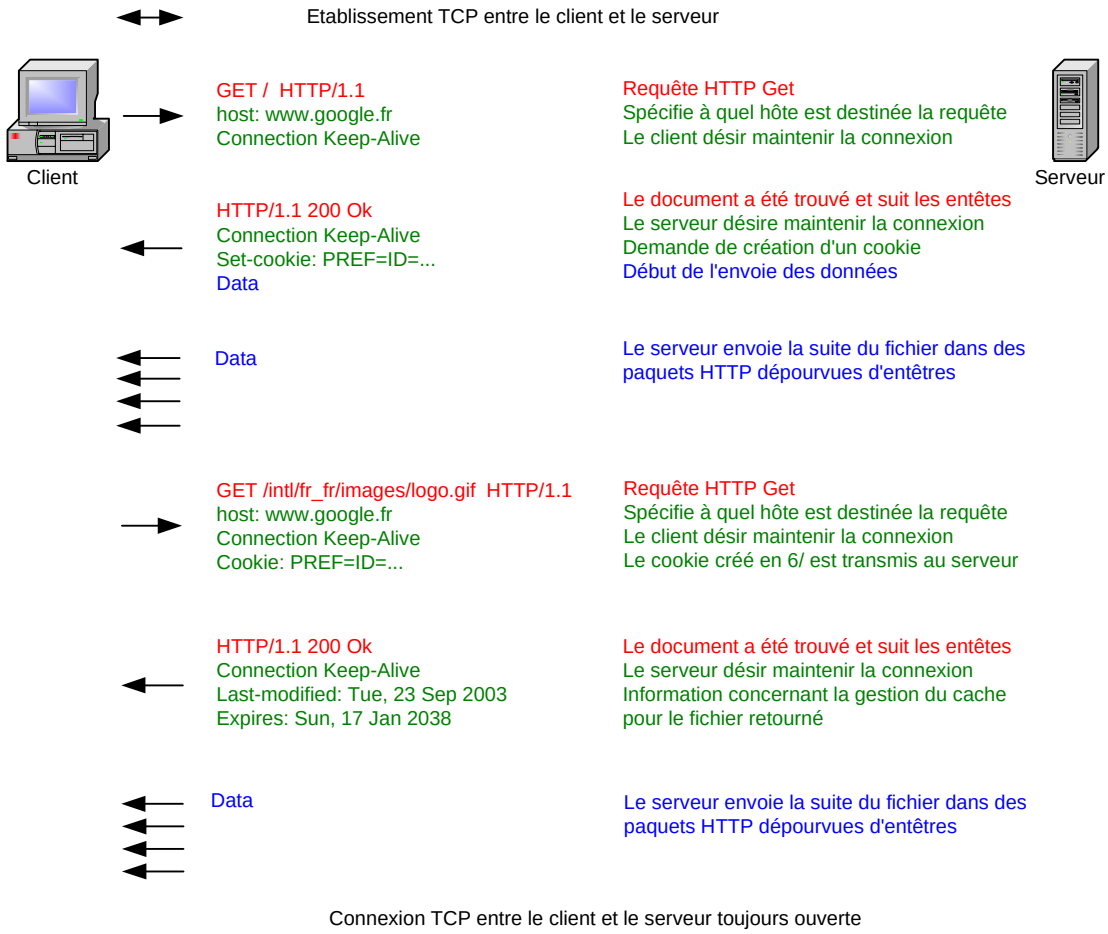
Lorsque le serveur renvoie un code de statut de la série 3xx, il accompagne sa réponse d'un en-tête *Location* indiquant la nouvelle adresse du document. Dans ce cas, le navigateur va automatiquement émettre une requête vers cette nouvelle adresse.




## 13 Analyse de protocole

L'acquisition a été effectuée à l'aide d'*Ethereal* (www.ethereal.com).

Scénario : Un client effectue une requête sur le serveur www.google.fr

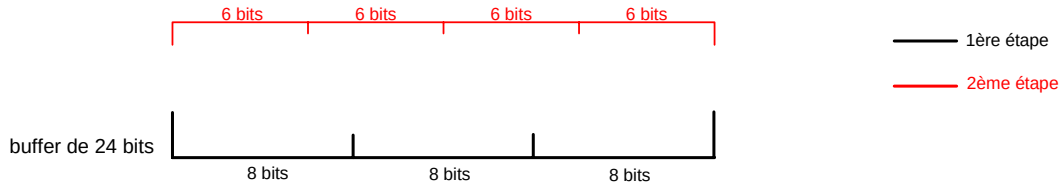


# 14 Encodage base 64

## 14.1.1 Quand l'encodage base 64 est-il utilisé ?

L'encodage en base 64 est principalement utilisé pour la transmission des fichiers binaires (images, exécutables etc…) par *e-mail*.
Il est également utilisé pour masquer le mot de passe transmis lors de l'authentification *Basic* du protocole HTTP.
Cet encodage ne consiste en aucun cas en un chiffrement des données. Il ne fournit qu'une autre manière de représenter les données.

## 14.1.2 Principe




Chaîne de conversion : ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

Valeurs d'Index : A=0 , B=1 , ... , W=22 , Z=25 , l=37, n=39 , ... , 9=61 , +=62 , /=63

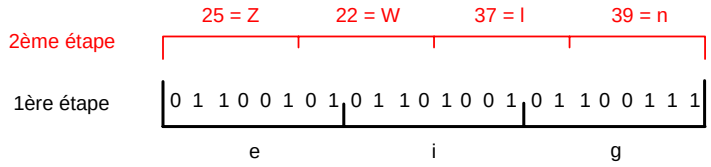
**1ère étape** : remplir le *buffer*
Le premier octet à encoder est placé dans les 8 bits de poids forts d'un *buffer* de 24 bits. Le deuxième octet est placé dans les 8 bits suivants et le troisième octet dans les 8 bits de poids faibles. Dans le cas où moins de 3 octets sont à encoder, les bits vides du *buffer* sont remplis avec des 0.

**2ème étape** : Conversion
Les 6 bits de poids forts du *buffer* sont pris comme index dans la chaîne de conversion « ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ ». Le caractère désigné par l'index ainsi établi est le premier caractère converti en base 64.
Les 6 bits suivants sont pris comme index dans la chaîne de conversion. Le caractère désigné par l'index ainsi établi est le deuxième caractère résultant de la conversion.
De même pour les 2*6 bits restants.

Remarque :
Si moins de 3 octets sont à encoder, cela ce traduit par le remplissage du *buffer* par des 0. Les groupes de 6 bits ainsi remplis sont convertis en « = ».

## 14.1.3 Exemple : Conversion de « eig »

| **Caractère** | e | i | g |
|---|---|---|---|
| **ASCII** (décimal) | 101 | 105 | 103 |
| **8 Bits** (binaire) | 01100101 | 01101001 | 01100111 |




Après conversion au format base 64, « eig » devient « ZWln »

## 14.1.4 Failles liées à l'encodage en base 64 :

Après conversion en base 64, les seuls caractères possibles sont ceux de la chaîne de caractères « ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ ».

C'est en se basant sur cette constatation que les failles liées à l'encodage base 64 sont apparues.

Forts de cette constatation, certains développeurs n'ont pas intégrés à leurs codes une gestion d'erreurs dans le cas où un caractère autre que ceux de la chaîne de caractères Base 64 se présenterait.

#### 14.1.4.1 Exploit d'une faille liée à l'encodage Base 64

**Hypothèses :**
Un script s'appuyant sur l'authentification *Basic* du protocole HTTP est utilisé pour authentifier les clients.
Le navigateur se connecte en passant par un *proxy* local (*WebScarab* [9] par exemple) afin de pouvoir modifier la requête.

**Déroulement du scénario :**
1/ Le navigateur va afficher une fenêtre demandant à l'utilisateur d'entrer ses données d'authentification (*Username/Password*)
2/ Une fois les données entrées, le navigateur les convertit au format base 64 et les envoie au *proxy*.
3/ Les données d'authentification encodées en base 64 sont modifiées sur le *proxy* de façon à y insérer un caractère autre que ceux générés par un encodage base 64
4/ Le *proxy* transmet la requête modifiée au serveur Web
5/ Le serveur Web reçoit les données d'authentification et les passe au script chargé de valider ou d'invalider l'authentification.
6/ Le script va être confronté à un cas non prévu et va planter

Ce plantage va se traduire, selon la configuration matérielle et logicielle du serveur, par un *crash* du serveur, un gain de privilèges etc…

## 15 Conclusion

A partir d'une norme minimale HTTP 1.0, diverses évolutions ont permis de définir la version 1.1 du protocole HTTP améliorant aussi bien les performance (connexions persistantes § 7) que la sécurité (*Digest Authentication* § 11.2) et les mécanismes déjà existants (cache § 8, négociation de contenu § 9, Cookies § 10 …).

Cependant, de nouvelles fonctionnalités (Active X, applets Java, javascript, *e-banking*, *webmail*, administration d'un serveur via HTTP, SOAP, XML, …) sont ajoutées tous les jours aux possibilités offertes par les navigateurs et serveurs Web, et chacune d'elles est une nouvelle source potentielle de failles.
Toutes ces applications utilisent le port 80 et le protocole HTTP pour communiquer.

Le niveau de complexité, en terme de nombre de fonctionnalités (*plugins*), atteint par les navigateurs et les serveurs Web est tel que ces applications (*IE, Netscape, IIS, Apache*…) possèdent toutes de nombreuses failles.
Ces failles se situent pour la plupart au niveau applicatif et peuvent être exploitées par un mauvais contrôle des paramètres reçus par un script (*Cross Site Scripting*, *Remote Directory Traversal*…), des *buffers overflow* (visant, par exemple, *Windows Media Player*, *Internet Explorer* ou même des processus système comme lsass.exe).

De plus, la démocratisation du *firewall* (souvent de type *Personal Firewall* pour les particuliers) laisse croire aux utilisateurs que leur poste est sécurisé et exempt de tout risque.
Malheureusement, afin de permettre la navigation sur le Web, les *firewalls* laissent généralement le port 80 ouvert et donc, une entrée directe (..., P2P, IM, ...) sur le poste client.
Les failles ne se situent pas au niveau du protocole HTTP mais au niveau des données transmises (cf. § 14.1.4 faille base_64).

## 16 Sources


- RFC 1945 "Hypertext Transfer Protocol – HTTP/1.0" [1]
  http://www.faqs.org/rfcs/rfc1945.html
- RFC 2616 "Hypertext Transfer Protocol – HTTP/1.1" [2]
  http://www.faqs.org/rfcs/rfc2616.html
- RFC 2617 "HTTP Authentication: Basic and Digest Access Authentication" [3]
  http://www.faqs.org/rfcs/rfc2617.html
- RFC 2965 "HTTP State Management Mechanism" [4]
  http://www.faqs.org/rfcs/rfc2965.html
- RFC 2396 "Uniform Resource Identifiers (URI): Generic Syntax [5]
  http://www.faqs.org/rfcs/rfc2396.html
- "Caching tutorial for web authors and webmasters" [6]
  http://www.mnot.net/cache_docs/
- "Négociation de contenu" [7]
  http://www.apachefrance.com/Manuels/Apache_1.3_VF/content-negotiation.html
- Sécurisation du trafic Web grâce à un proxy : L'exemple de Blue Coat [8]
  http://www.td.unige.ch/wsh/blue_coat.pdf
- Site du proxy WebScarab [9]
  http://www.owasp.org/software/webscarab.html