

# Cryptographie

Protocoles cryptographiques

Alexander Schaub<sup>1</sup>

[schaub.alexander@free.fr](mailto:schaub.alexander@free.fr)

16/12/2024

---

<sup>1</sup>DGA-MI, Bruz

## Dans l'épisode précédent...

- On a vu comment **partager** des clés entre deux participants
- On a vu comment **garantir l'authenticité** de données reçues
  - ... à condition d'avoir confiance en la **clé publique** de l'expéditeur !

Aujourd'hui, nous allons compléter ces services afin de construire des protocoles de la vraie vie :

## Dans l'épisode précédent...

- On a vu comment **partager** des clés entre deux participants
- On a vu comment **garantir l'authenticité** de données reçues
  - ... à condition d'avoir confiance en la **clé publique** de l'expéditeur !

Aujourd'hui, nous allons compléter ces services afin de construire des protocoles de la vraie vie :


- SSH v2.0 (RFC4253)
- TLS v1.3 (RFC8446)

# Authentification symétrique

Mais avant cela, petit interlude !

- Lors de la dernière séance, nous avons vu l'authentification **asymétrique** grâce aux **signatures électroniques**
- L'authentification symétrique existe en deux variantes :
  - le MAC
  - le chiffrement authentifié (ou AEAD)

# MAC (Message Authentication Code)

- Rien à voir avec les Mac(intosh) de la marque à la 
- Rien à voir avec les adresses MAC (Media Access Control) Ethernet
- Comme une signature, mais avec la **même clé** de signature et de vérification

## MAC : notation

- Trois ensembles  $E$  (messages),  $F$  ( $tags \approx$  signatures),  $K$  (clés MAC)
- De deux fonctions,  $f : E \times K \mapsto F$ ,  $g : F \times E \times K \mapsto \{\text{true}, \text{false}\}$  telles que
  - $\forall x \in E, k \in K, g(f(x, k), x, k) = \text{true}$
  - Trouver  $y \in F$  tel que  $g(y, x, k) = \text{true}$  revient à connaître  $k$

## MAC : construction HMAC (1)

- Les fonctions hachage semblent obtenir des bonnes propriétés pour construire un MAC
- Comment combiner message et clé ?
  - Première tentative :  $f(x, k) = \text{HASH}(k \parallel x)$ 
    - N'est pas sûr ! MD5, SHA1 et SHA2 sont susceptibles à des attaques **par extension de longueur**.

Étant donné  $\text{HASH}(m)$  et la longueur de  $m$ , on peut calculer  $\text{HASH}(m \parallel m')$  pour  $m'$  arbitraire (L'état interne de ces fonctions de hash à la fin du message est connaissable à partir du haché produit)

## MAC : construction HMAC (2)

- Deuxième tentative :  $f(x, k) = \text{HASH}(x \parallel k)$ 
  - Mieux mais à cause de l'attaque par extension de longueur, une collision de HASH sur  $x$  produirait une collision sur  $f(x, k)$ .
- Bonne solution :  $f(x, k) = \text{HASH}(k \parallel \text{HASH}(k \parallel x))$ 
  - Encore mieux si on n'utilise pas la même clé les deux fois :

$f(x, k) = \text{HASH}(k \oplus \text{opad} \parallel \text{HASH}(k \oplus \text{ipad} \parallel x))$  avec IPAD, OPAD deux constantes (RFC2104)



## MAC : à quoi ça sert ?

Imaginons que vous envoyez ce message à votre banquier :

Envoyez 10000 euros sur le compte FR76123456789. Mot de passe convenu : bqs87DQ9sdf198qsf

Ce message est chiffré avec un **chiffrement par flot** (rappel : consiste à générer une séquence pseudo-aléatoire et à XORer avec le message).

Eve intercepte le message, elle connaît l'ordre et sa formulation mais pas le mot de passe. Elle veut détourner l'argent sur son compte. Comment s'y prend-elle ? Comment modifie-t-elle le message ?

# Chiffrement authentifié

- On veut souvent protéger la confidentialité **et** l'intégrité des données
- Combiner les deux services = chiffrement authentifié (ou AEAD)  
L'AEAD permet aussi d'authentifier des données claires en plus mais c'est du bonus
- En général, consiste à combiner un **mode de chiffrement** avec un algorithme de **MAC**
- Algorithmes les plus utilisés :
  - **AES-GCM** : AES-CTR + GHASH
  - **ChaCha20Poly1305** : ChaCha20 + Poly1305
- Constructions plus anciennes : chiffrement + HMAC avec deux clés **distinctes**

# AES-GCM

- Utilisé avec de clés de 128-bits (AES-128) et IV de **96 bits**
  - Valeur potentiellement problématique : l'espace d'IV est petit si généré aléatoirement
  - L'IV ne doit **jamais** être réutilisé (fuite de la clé d'authentification)

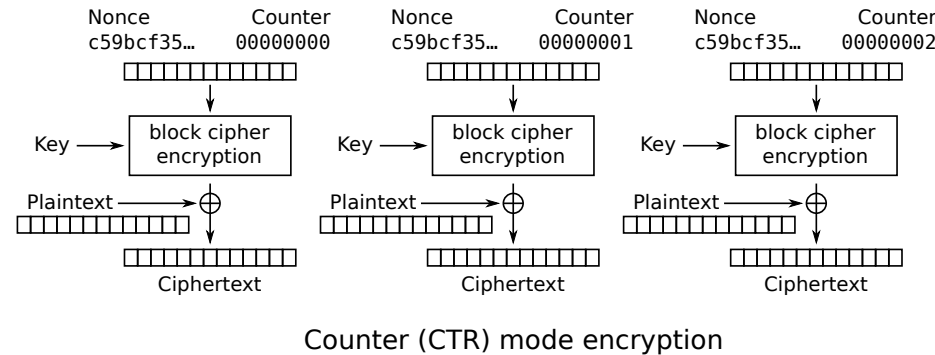


Image 1: (Dé-)Chiffrement CTR

# GHASH

Évaluation de polynôme : si  $S_i \in (0, 1)^{128}, i = 1..m + n + 1$  sont les données à protéger en intégrité ( $m$  blocs de données additionnelles,  $n$  blocs de chiffré paddés avec des 0 + les tailles), alors

$$\text{tag} = \text{Chiff}_k(\text{IV} \parallel 0^{32}) + \sum_{i=1}^{m+n+1} S_i \cdot H^{2+m+n-i}$$

[dans  $\text{GF}(2^{128})$ ]

où  $H = \text{Chiff}_k(0^{128})$

# Chacha20Poly1305

- Clé de 256 bits, IV de 96 bits
  - XChacha20Poly1305 avec IV de 192 existe et serait mieux, mais pas standardisé 🙄
- Poly1305 : évaluation de polynôme. Soit  $S$  identique que précédemment,

$$\text{tag} = \left( s + \sum_{i=1}^{m+n+1} (S_i + 2^{128}) \cdot r^{2+m+n-i} \bmod (2^{130} - 5) \right) \bmod (2^{128})$$

où  $\text{Chacha20}_k(\text{IV} \| 0^{32}) = r \| s \| 0^{256}$  et  $|r| = |s| = 128$  bits

(pas tout à fait exact pour le dernier  $S_i$  mais passons)

# Stockage de mot de passe

Email	Mot de passe
xxkevindu36xx@hotmail.fr	roxxor
jean.charles@gmail.com	pupuce1993
john.doe@fr.fr	lkqj098qlkjdq78!09dq
xxxamandinedu39xxx@skyblog.fr	roxxor

→ Il ne faut jamais stocker les mots de passe en clair

Risque de fuite de données, mise en danger des utilisateurs si réutilisation du mot de passe, etc.

# Stockage des mots de passe : fonction de hachage

Email	Mot de passe
xxkevindu36xx@hotmail.fr	342887f489f...
jean.charles@gmail.com	94363ba85ef0f...
john.doe@fr.fr	ced2f800c497ba2...
xxxamandinedu39xxx@skyblog.fr	342887f489f...

## Deux problèmes :

- Mots de passe identiques = hachés identiques
- Haché fonction du mot de passe uniquement : possibilité d'attaque par dictionnaire inversé

## Stockage des mots de passe : fonction de hachage + sel

Email	sel	Mot de passe
xxkevindu36xx@hotmail.fr	su8qlsdu	34e3808164a...
jean.charles@gmail.com	od9us67s	7c27b2a16d27429...
john.doe@fr.fr	nsk9jd24	34a37ab14b39dafd...
xxxamandinedu39xxx@skyblog.fr	nmqh75s0	c8d6c66cebe42...

### Un problème :

- Mots de passe identiques = hachés différents , mais
- Fonction de hachage facile à calculer : inversion d'un mot de passe faible possible



# Stockage de mots de passe : dérivation de clé

Utilisation similaire à fonction de hachage avec sel mais :

- optimisé pour être difficile à inverser par GPU / puce dédiée (ASIC)
- Malheureusement, possèdent souvent plein de paramètres à choisir (prendre ceux par défaut de la bibliothèque cryptographique choisie !)

Algorithmes conseillés :

- argon2id
- balloon
- scrypt

Algorithmes historiques :

- PBKDF1 (à éviter !)
- PBKDF2 (HMAC itéré)
- bcrypt

# Protocoles cryptographiques

- On a vu tous les “blocs de base” de la cryptographie “courante”
- Il faut ensuite les associer pour obtenir des services possédant les bonnes propriétés :
  - confidentialité des échanges
  - intégrité des données échangées
  - garanties quant à l’identité des participants
  - ...

Nous en verrons deux plus en détail: SSHv2 et TLS v1.3.

# SSH : présentation

Permet de se connecter **de façon sécurisée** à un serveur distant et d'agir comme si on y était connecté physiquement (entre autres; permet aussi le transfert de fichier, le *tunneling* d'autres flux réseau, etc.)

Protocole assez ancien : première version en 1995, pour remplacer telnet. v2 améliorant grandement la sécurité en 2006.

# SSH : Négotiation d'algorithmes



Liste d'algorithmes acceptés

Algorithmes pour :

- échange de clés
- capacité des clés serveur (signature ou chiffrement)
- chiffrement de données client  $\rightarrow$  serveur et serveur  $\rightarrow$  client
- MAC client  $\rightarrow$  serveur et serveur  $\rightarrow$  client

# SSH : Négotiation d'algorithmes



Liste d'algorithmes acceptés

Algorithmes pour :

- échange de clés
- capacité des clés serveur (signature ou chiffrement)
- chiffrement de données client  $\rightarrow$  serveur et serveur  $\rightarrow$  client
- MAC client  $\rightarrow$  serveur et serveur  $\rightarrow$  client

# SSH : Échange de clés (Diffie-Hellman)



$x$

$$e = g^x \bmod p$$



Note: le groupe (défini par  $g$  et  $p$ ) est public et partagé suite à la négociation d'algorithmes

# SSH : Échange de clés (Diffie-Hellman) (II)



$x$

$$\overleftarrow{K_{\text{verif}}, g^y \bmod p, \text{Sign}_{K_{\text{sig}}}(H)}$$



$$K_{\text{sign}}, K = e^y \bmod p,$$

$$H =$$

$$\text{Hash}(K \parallel e \parallel f \parallel \dots)$$

Deux manières de vérifier l'authenticité de  $K_{\text{verif}}$ :

- Un certificat (plus d'infos dans la prochaine partie !)
- TOFU (*Trust On First Use*) : on accepte la première clé publique reçue du serveur et on la stocke pour une vérification ultérieure

## SSH : Échange de clés (Diffie-Hellman) (II)



$$x, K = f^x \bmod p, H = \text{Hash}(\dots)$$



$$K_{\text{sign}}, K = e^y \bmod p, \\ H = \text{Hash}(K \parallel e \parallel f \parallel \dots)$$

Le client vérifie la signature. Si elle convient, le serveur et le client se sont mis d'accord sur  $H$  et  $K$ , puis dérivent :

- l'IV de chiffrement initial (client  $\rightarrow$  serveur et serveur  $\rightarrow$  client )
- la clé de chiffrement et de MAC client  $\rightarrow$  serveur et serveur  $\rightarrow$  client



## SSH : Tunnel non-sécurisé

Un paquet SSH échangé contient les données suivantes :

- taille du paquet (sur 4 octets)
- taille du padding (sur 1 octet)
- données utiles (taille du paquet - taille du padding - 1)
- padding aléatoire (taille nécessaire)
- MAC (taille en fonction de l'algo MAC choisi)

Padding : au moins 4 octets, la taille des quatre premières données doit être multiple de 8 ou de la taille du bloc de l'algorithme de chiffrement.

## SSH : Tunnel sécurisé

Après échangé des clés :

- les quatre premières données sont chiffrées (taille du paquet, taille du padding, données utiles, padding aléatoire)
- le MAC est calculé sur ces quatre champs en clair concaténé à un numéro de séquence (pour éviter le rejeu)

Le tunnel sécurisé n'est pas particulièrement élégant - pouvez-vous me dire pourquoi ?

## SSH : authentication

Une fois le tunnel établi, le client peut se connecter :

- soit via clé publique : le serveur connaît la clé publique, et le client signe (entre autres) la valeur  $H$
- soit via mot de passe, en le transmettant dans le tunnel sécurisé

## TLS 1.3 : Le HTTPS *moderne*

Un petit récapitulatif :

- SSL 1.0 initialement développé par Taher ElGamal (le même qui a inventé le chiffrement ElGamal) chez Netscape (l'ancêtre de Mozilla)
- SSL 1.0 est tout cassé, Netscape développe SSL 2.0 qui est rendu public en 1995
- SSL 2.0 est **également** tout cassé, ce qui donne SSL 3.0 en 1996
- L'IETF se réveille en 1999 et standardise TLS1.0, une variante de SSL 3.0
- TLS 1.1 et 1.2 modernisent *un peu* le protocole
- TLS 1.3, sorti en 2018, fait le grand ménage, enlève les options obsolètes et corrige plusieurs failles

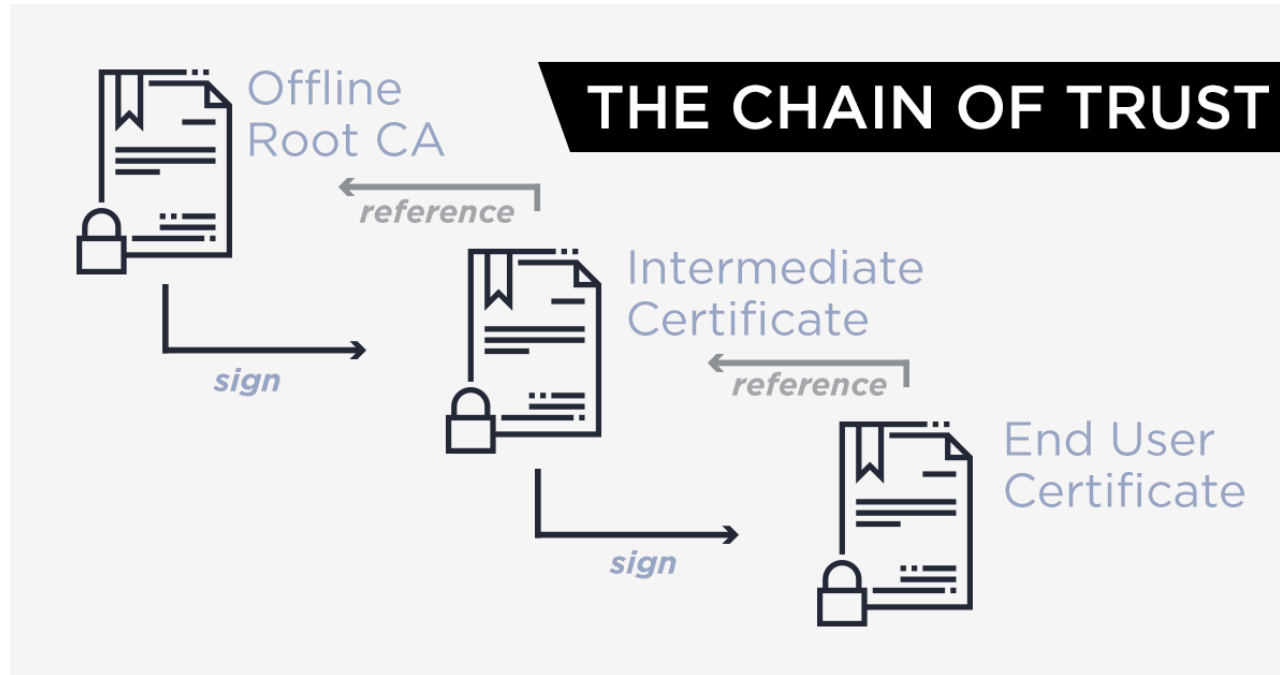
## TLS : Objectif

Il s'agit d'établir **un tunnel sécurisé** entre votre **navigateur** et un **serveur web** (proposant un service HTTP).

Doit être compatible HTTP : le protocole fonctionne au-dessus d'une connexion **TCP** uniquement.

Doit être résistant aux attaques par homme-du-milieu : plus question de *TOFU*, mais de **certificats**.

# TLS : Certificats



source: <https://www.exoscale.com>

## Aparté : Que contient un certificat ?

- Dates de validité (pas avant / pas après)
- Nom(s) de domaine
  - avec potentiellement des *wildcard*, i.e. tous les sous-domaines
    - par contre, \*.com est interdit. Et \*.co.uk aussi...
- Une clé publique (pour la signature, ou le chiffrement asymétrique, ou l'échange de clés)
- Les usages possibles (autorité racine, intermédiaire, certificat final)
- Une signature d'une autorité supérieure
- L'identifiant de cette autorité

# TLS1.3 : Négotiation d'algorithmes



—————→  
Liste d'algorithmes acceptés



Algorithmes pour :

- chiffrement authentifié/dérivation de clé HKDF
- groupes (EC)DHE supportés + clés publiques associées
- algorithmes de signature (pour TLS / optionnellement pour certificats)
- *optionnellement : clé pré-partagée (issue d'une connexion précédente)*



# TLS1.3 : Choix d'algorithmes



← Choix d'algorithme



- Choix d'algorithme de chiffrement
- Choix de groupe (EC)DHE + clé publique éphémère associée

## TLS1.3 : Certificat serveur



←  
Certificat + Signature



- certificat (contenant une clé de vérification) + chaîne de certificats
- signature du message client + message serveur + certificat (avec la clé correspondante au certificat)

## Aparté : Diffie-Hellman éphémère

Pour faire un échange de clés authentifié :

- Soit le certificat contient la clé publique DH du serveur,
- Soit :
  - le certificat contient une clé de vérification de signature,
  - le serveur génère un nouveau bi-clé pour chaque échange de clés
  - le serveur signe la clé publique de cette bi-clé

Deuxième solution appelée “échange de clé éphémère” (la première est dite *statique*)

## Aparté : Diffie-Hellman éphémère et *PFS*

PFS : *Perfect Forward Secrecy*

Même si la clé **privée** du certificat serveur est diffusée, les échanges **antérieurs à cette diffusion** ne peuvent pas être compromis.

Diffie-Hellman éphémère possède la propriété PFS. Ce n'est **pas le cas** pour un Diffie-Hellman statique.

## TLS1.3 : Fin de la négociation serveur



←  
HMAC(...)



Le serveur envoie un HMAC des messages précédents. La clé est dérivée du secret partagé par (EC)DHE.

## TLS1.3 : Fin de la négociation client



Le **client** envoie un HMAC des messages précédents. La clé est dérivée du secret partagé par (EC)DHE. Cela assure que les deux parties ont bien **dérivé la même clé**.

## TLS1.3 : Tunnel sécurisé

Après établissement d'une clé de session :

- Les données sont envoyées chiffrées en utilisant du **chiffrement authentique**
- L'en-tête clair contient la taille des données
- Les applications utilisant TLS peuvent utiliser un **padding optionnel** (qui sera chiffré) pour tenter de **cacher la taille des données**

La semaine prochaine : un petit bonus et  
**présentation des projets**

Bon courage !