

# PROJET SR2I208

## Analyse et expérimentation de TLS 1.3

*OUZINEB Sohaïb-ASSOMANY Marvin-PROIETTI Harith*  
*Supervisé par : Ahmed Serhrouchni*

### Sommaire:

0 - Introduction - Evolution historique

I - Fonctionnement de TLS

1) Handshake TLS1.2

2) TLS 1.3

a) De TLS 1.2 à TLS 1.3

b) Initialisation de session (4 étapes)

c) Reprise de connexion (TLS 1.2 et TLS 1.3)

d) 0-RTT

3) Gestion des services cryptographiques

a) Gestion de l'authentification

b) Vérification des certificats

c) Gestion de la confidentialité

d) Gestion de l'intégrité

e) Gestion du non-rejeu

II - Benchmarking

1) Introduction

2) Temps de connexion

3) Modélisation de plusieurs clients

III - Extensions Client Hello

1) Extension replay protection

2) Extension rejected reason

Conclusion

Bibliographie

## Résumé:

Nous analysons dans ce rapport les apports de TLS 1.3 par rapport à TLS 1.2 en analysant de manière précise le handshake d'initialisation de session, de reprise de session et de 0-RTT en mettant en valeur le rôle de chaque champ.

Nous proposons par la suite un benchmarking à échelle moyenne permettant d'évaluer le temps de connexion et de le comparer à TLS 1.2. Puis, nous modélisons plusieurs clients effectuant plusieurs requêtes à travers différents streams pour tester les performances à échelle moyenne.

Enfin, nous analysons les faiblesses du 0-RTT et proposons des extensions au client Hello pour y remédier: une pour indiquer la politique anti-replay et une autre pour indiquer la valeur associée à cette politique. Nous proposons aussi une autre extension pour que le client puisse connaître la cause de rejet de 0-RTT pour empêcher une attaque replay plus générale.

## Introduction:

TLS (Transport Layer Security) est un protocole de chiffrement et de sécurisation des échanges sur Internet, successeur du protocole désormais obsolète SSL (Secure Socket Layer). SSL, qui fut créé en 1994 par l'organisation Netscape, donna lieu à 3 versions successives entre 1994 et 1997, avant que la troisième version SSL 3.0 soit reprise par l'IETF et remplacée en 1999 par TLS (Transport Layer Security).

Couramment, on utilise indifféremment la dénomination TLS, SSL ou encore SSL/TLS pour désigner ce protocole de chiffrement largement utilisation par les communications actuelles. Notons que TLS est assez largement inspiré de SSL 3.0, bien que des améliorations notamment au niveau de la sécurité y soient apportées.

Actuellement, la version de TLS utilisée est TLS 1.3, dont la RFC fut publiée en août 2018 . Cette dernière version marque notamment la disparition de certaines primitives cryptographiques jugées trop faibles en termes de sécurité (MD4, RC4, DSA ou SHA-224). Par ailleurs, le "handshake" devient mieux optimisé que dans les versions précédentes, et la configuration de TLS 1.3 est plus facile. On note également l'apparition de l'option "0-RTT", qui permet de démarrer immédiatement une communication entre un client et un serveur ayant déjà communiqué auparavant.

Le clair avantage de SSL/TLS est son ergonomie : en effet, il s'intègre très bien aux protocoles couramment utilisés sur le net, notamment HTTP, qui n'a besoin que d'être très peu modifié pour permettre d'établir la connexion avec TLS. Ce dernier permet de faire la connexion entre la couche transport et la couche applicative. Couramment, on considère qu'il s'intègre au niveau session (niveau 5) du modèle OSI. La fusion entre HTTP et SSL donne lieu au protocole bien connu HTTPS, fonctionnant sur le port 443.

Actuellement, TLS 1.3 est utilisé sur de nombreux serveurs (Apache, NGINX...) et browsers (Chrome, Safari, Opera...), son fonctionnement en faisant un protocole optimisé tout en étant sécurisé.

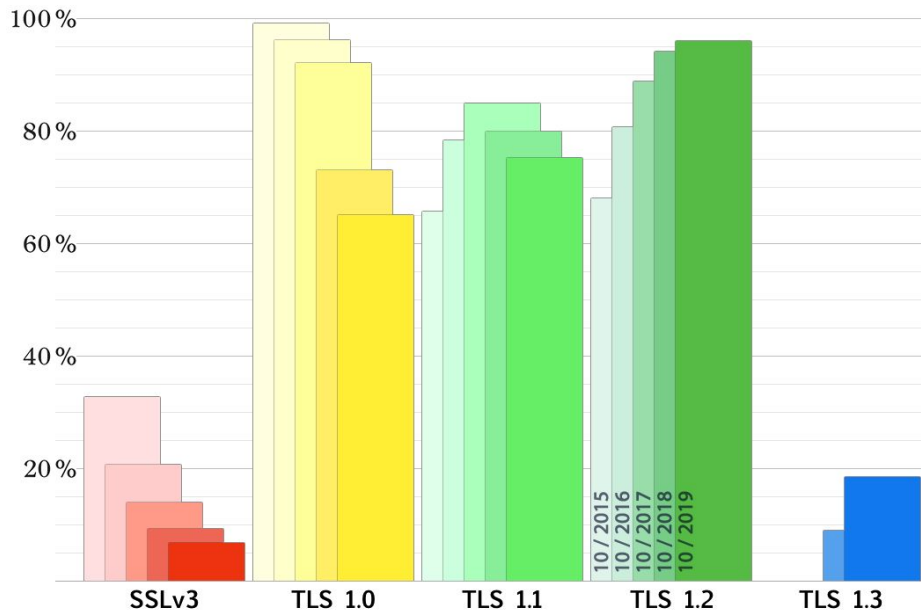


Figure : Evolution des versions de SSL/TLS acceptées par les serveurs scrutés par SSL Pulse sur la période 2015-2018

## I Fonctionnement de TLS

### 1) Handshake jusqu'à TLS 1.2

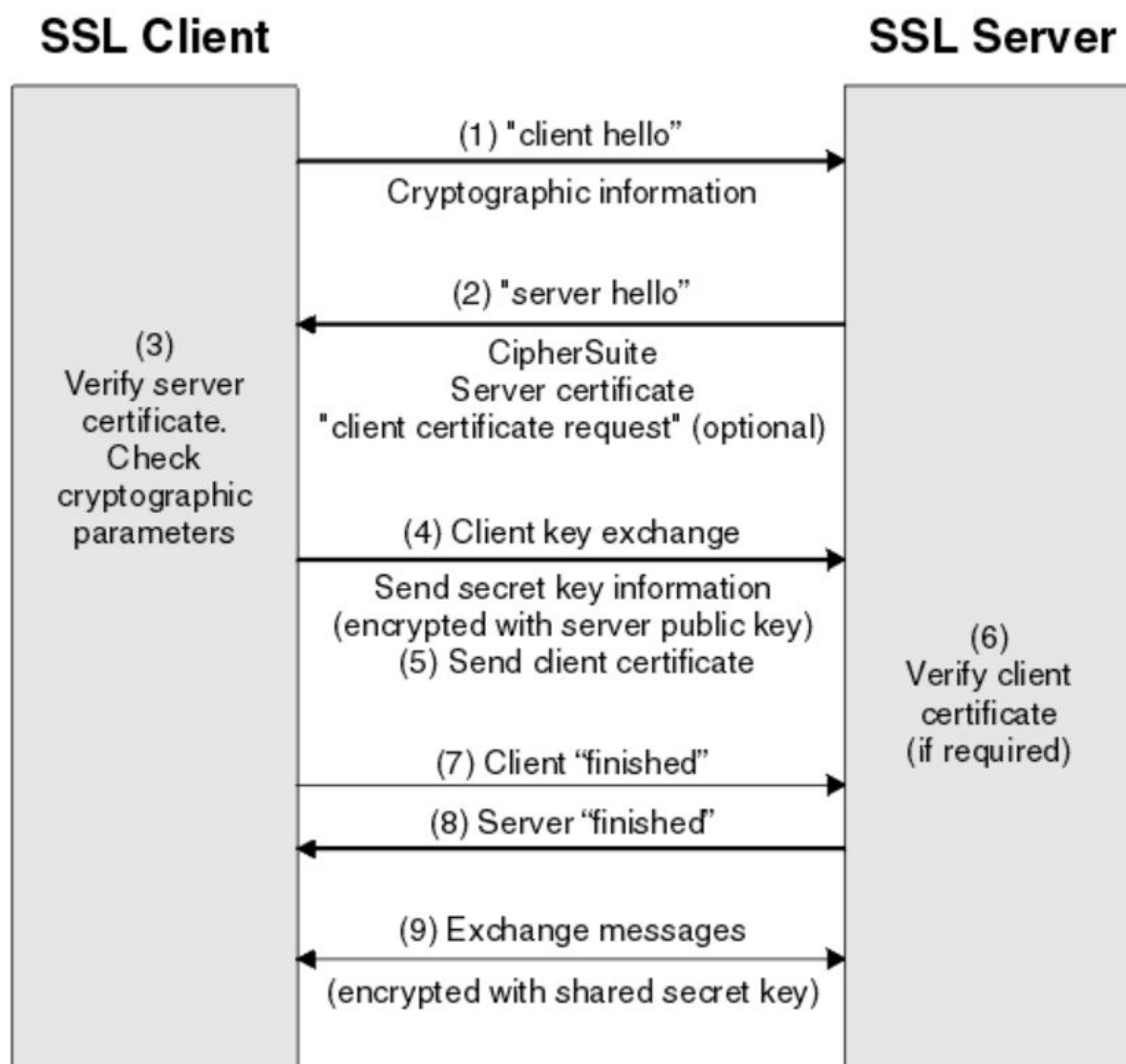
Le protocole TLS est fondé sur une architecture de type client-serveur. Lors de l'établissement de la connexion, le client et le serveur s'envoient à tour de rôle des requêtes accompagnées d'informations (certificats, clés secrètes partagées...) dans une étape nécessaire appelée "handshake". Cette initiation de la communication va permettre de s'assurer que les principaux services cryptographiques vont être respectés au cours des échanges futurs. Les principaux mécanismes devant être respectés sont l'authentification, la confidentialité, l'intégrité ainsi que le non-rejeu.

Nous allons dans un premier temps présenter le handshake TLS 1.2, puis le comparer avec son successeur TLS 1.3 afin de mieux comprendre les corrections et améliorations apportées.

L'initiation d'une communication se déroule de la manière suivante :

- 1) Le client et le serveur s'accordent sur la version du protocole à utiliser
- 2) L'algorithme cryptographique lors des communications futures est choisi
- 3) Le client et le serveur s'authentifient réciproquement en s'échangeant puis validant des certificats digitaux.
- 4) Un chiffrement asymétrique est utilisé pour générer et partager la clé secrète partagée, afin de sécuriser sa transmission
- 5) Une fois cette clé partagée, le client et le serveur peuvent désormais communiquer en chiffrant leurs messages avec la clé partagée en 4). Le chiffrement est donc cette fois symétrique, ce qui est plus rapide qu'un chiffrement asymétrique.

Voici un schéma détaillé montrant comment se déroule un handshake utilisant ce protocole :



Handshake jusqu'à la version TLS 1.2 incluse

- Etape 1 : Le client SSL/TLS envoie un message “client hello” au serveur, dans lequel il liste des informations comme la version de SSL/TLS qu’il supporte ainsi que la listes des suites cryptographiques (ensemble des algorithmes de chiffrement supportés) par ordre de préférence, du plus robuste au plus faible. Il envoie aussi les méthodes de compression qu’il supporte. Par ailleurs, il envoie une chaîne d’octets aléatoires (“Client hello”) qui va permettre d’éviter une attaque type “man in the middle” (cf étape 2)

```

Ciphersuites (34 suites)
Cipher suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
Cipher suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
Cipher suite: TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x0088)
Cipher suite: TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA (0x0087)
Cipher suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
Cipher suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
Cipher suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
Cipher suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
Cipher suite: TLS_RSA_WITH_CAMELLIA_256_CBC_SHA (0x0084)
Cipher suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
Cipher suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
Cipher suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
Cipher suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
Cipher suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
Cipher suite: TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x0045)
Cipher suite: TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA (0x0044)

```

#### Exemple de Cypher Suites proposées lors d’un “Client hello”

- Etape 2 : Le serveur répond avec un “serveur hello” qui contient la suite cryptographique qu’il a choisie parmi celles proposées par le client, ainsi que l’identifiant de session (session ID) et une chaîne aléatoire de 32 octets (“Server hello”). Ces chaînes aléatoires sont donc importantes car elles vont permettre de vérifier que le client parle bien au serveur et réciproquement, comme nous l’expliquerons plus loin.

Le serveur envoie aussi son certificat digitalisé (X.509), et peut en demander un de la part du client pour qu’il s’authentifie en envoyant une “requête de certificat du client” (“certificate request”), dans laquelle il précise les format de certificats qu’il supporte. Il envoie également sa clé publique au client, qui sera nécessaire pour l’échange des clés (“server key exchange”). Il envoie enfin un “server Hello Done” indiquant que le ServerHello et les messages associés sont terminés.

- Étapes 3, 4 et 5 : Le client vérifie le certificat digitalisé envoyé par le serveur. Si le serveur a envoyé une requête de certificat du client, le client envoie également une chaîne d’octets aléatoire chiffrée avec sa clé privée et contenant son certificat digital. (ou une “no digital certificate alert”). Dans certains handshakes, l’absence de cette authentification du client va empêcher l’établissement de la communication. Par ailleurs, le client envoie une autre chaîne d’octets aléatoires (“pre master secret”) qui

va servir, combinée au “client random” et “server random” (étapes 1 et 2), à générer le “master secret” qui sera la clé symétrique utilisée lors des communications futures. Cette chaîne “pre secret master” est chiffrée avec la clé publique du serveur. Toute cette étape est appelée “client key exchange”.

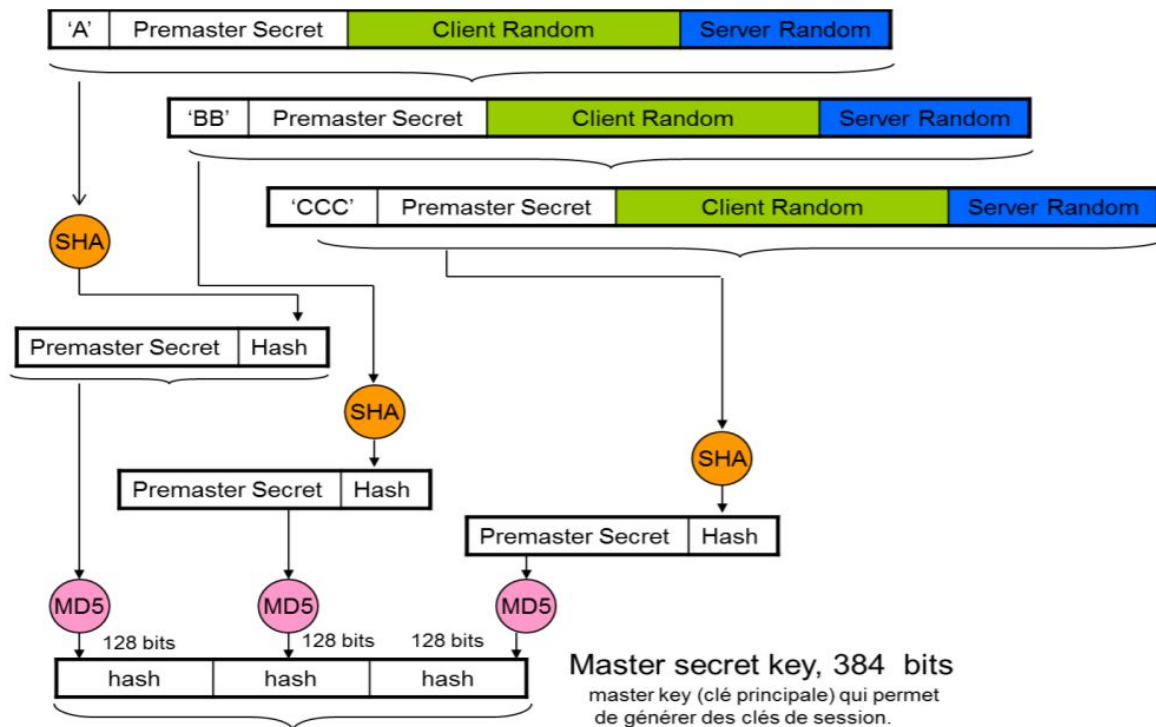
On comprend alors le rôle des “random client” et “random server” au début. S’ils n’avaient pas eu lieu, toute la génération de la clé finale secrète serait revenue au client, qui envoie le “pre master secret”. Ainsi, si le serveur générait quant à lui des messages avec un chiffrement fixé par le client, dans une attaque type “man in the middle”, on pourrait imaginer un attaquant qui récolte la clé “pre master secret” envoyée par le client et se fait passer pour le serveur sans que cela soit détecté par le client (attaque “replay”). Pour cela, il est important qu’il y ait une construction de la clé basée des deux côtés, et différente à chaque communication : on parle de “génération d’aléa”. Si le serveur envoie une chaîne aléatoire au client au début, ce dernier peut vérifier par la suite que c’est bien le même serveur qui avait envoyé la chaîne random au début, puisque celle-ci est utilisée dans la conception de la clé finale “secret master” et que par ailleurs seul le serveur pouvait décoder le “pre master secret” avec sa clé privée.

- Etape 6 : Le serveur SSL/TLS vérifie le certificat du client si demandé. Il décode le “pre master secret” avec sa clé privée (il est le seul à pouvoir le faire. Le master secret peut alors être calculé des deux côtés :

$$\text{master\_secret} = \text{PRF}(\text{pre\_master\_secret}, \text{"master secret"}, \text{ClientHello.random} + \text{ServerHello.random})$$

où PRF est une fonction pseudo-aléatoire.

Ci-dessous un schéma expliquant en détail le calcul de ce “Master secret” :



### Génération du Master secret

- Etape 7 : Le client envoie alors au serveur un message "final" (" client finished "), qui est chiffré avec la nouvelle clé secrète calculée comme décrit ci-dessus ("change cipher spec"). Avec cette étape, le client indique que le handshake est terminé de son côté, et pourra vérifier que le serveur a bien pu décoder le message avec leur clé secrète commune.
- Etape 8 : Le serveur envoie au client un message "final" (" server finished ") qui est aussi chiffré avec la nouvelle clé secrète commune ("change cipher spec"), et indique de cette manière que la handshake est également complété de son côté.
- Etape 9 : Pour la suite de la conversation, le client et le serveur peuvent désormais échanger des messages chiffrés de manière symétrique avec la clé secrète partagée.

Lors des prochaines connexions, de nouvelles clés pourront être dérivées à partir du "Master Secret" en utilisant par exemple la fonction de hachage MD5 combinée à de nouvelles chaînes aléatoires générées côté serveur et côté client. Notons que la session précédente est stockée en cache par le serveur, et pourra être retrouvée notamment à partir du Session\_ID (qui est initialisé à 0 lors de la première connexion)

On constate que le handshake a pris deux RTT (Round-Trip-Time) pour être complet. En général, on estime cette durée entre un quart et une demi seconde, mais cette durée peut varier selon d'autres facteurs. Quand on compare le TTFB (Time-To-First-Byte) entre HTTP et HTTPS, on constate que l'établissement de la connexion avec HTTPS est plus long, ce qui est normal.

## 2) TLS 1.3

### a) De TLS 1.2 à TLS 1.3

Deux problèmes de TLS 1.2 ont conduit à le faire évoluer : d'une part, le fait que l'initiation de toute connexion TLS 1.2 dure 2 allers-retours, en remarquant qu'en terme de chiffrement le premier aller-retour ne sert qu'à choisir la suite cryptographique sans que la construction du secret partagé (" master secret ") ne soit réellement avancée. Par ailleurs, la propriété de perfect forward secrecy n'est pas respectée avec des chiffrements de type RSA ou encore l'échange de clé Diffie-Hellman de type classique. En effet, dans ces types de chiffrement, si la clé secrète du serveur est compromise, tous les messages entre clients et serveurs vont pouvoir être décryptés.

C'est pourquoi TLS 1.3 est arrivé en force avec plusieurs mesures fortes :

- la suppression de toutes les suites cryptographiques jugées trop faibles en termes de sécurité. Un critère nécessaire pour qu'une suite cryptographique soit utilisée dans TLS 1.3 est qu'elle vérifie la propriété de PFS (Perfect Forward Secrecy), qui correspond au deuxième problème que nous avons soulevé avec TLS 1.2 . Ainsi, même la compromission d'une clé privée du serveur à un moment donné ne serait pas suffisante pour pouvoir retrouver toutes les clés utilisées de part et d'autre et ainsi décrypter totalement les communications. Le chiffrement RSA n'est donc bien sûr plus pris en charge, ainsi que DH statique. Par contre, l'échange de clé Diffie-Hellman dit " ephemeral " est supporté : il consiste à ce que les clés privées échangées lors d'une communication soient à usage unique. Ainsi, la compromission d'une clé à un instant donné ne permettra pas de décrypter toutes les communications antérieurs.

TLS 1.3 ne supporte finalement plus que 5 Ciphersuites :

- TLS\_AES\_128\_GCM\_SHA256
- TLS\_AES\_256\_GCM\_SHA384
- TLS\_CHACHA20\_POLY1305\_SHA256
- TLS\_AES\_128\_CCM\_SHA256
- TLS\_AES\_128\_CCM\_8\_SHA256



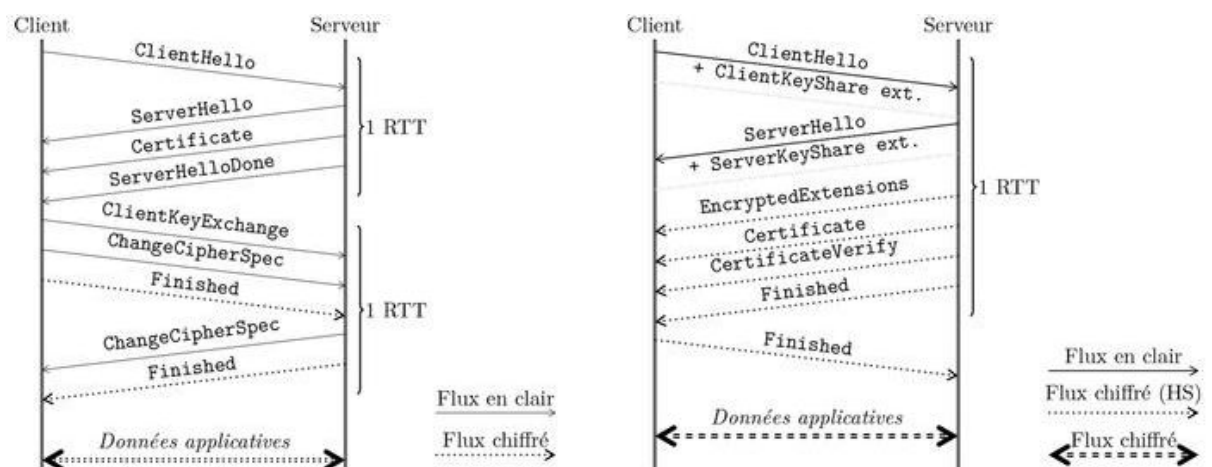
- la réduction de la durée du handshake, qui devient beaucoup plus optimisé en ne comportant qu'un seul aller-retour au lieu de deux pour TLS 1.2 . L'échange de clés et la définition de la primitive cryptographique utilisée pour les échanges se fait donc dès le premier handshake.
- Par ailleurs, on note également l'apparition de l'apparition " 0-RTT ", sur laquelle nous nous attarderons plus tard. Elle permet, entre un client et un serveur ayant déjà communiqué auparavant, d'envoyer des données applications dès le premier aller-retour.
- On peut mentionner que la compression a également été supprimée.

Nous allons tout d'abord nous attarder sur la nouvelle version du handshake version TLS 1.3, maintenant que l'on a vu le principe général d'un handshake TLS à travers TLS 1.2

- Le SNI (Server Name Indication) devient obligatoire pour TLS 1.3 : étant donné qu'un serveur peut héberger plusieurs noms de domaine, il convient pour un serveur lorsqu'il reçoit une requête de savoir quel nom de domaine est visé par le client. C'est le rôle du SNI, qui est une extension présente dans le Client Hello indication le nom de domaine du serveur avec lequel le client veut interagir.

## b) Handshake

Pour introduire le nouveau handshake, commençons par un schéma mettant en évidence la différence entre le handshake TLS 1.2 et TLS 1.3 .



A gauche figure le handshake TLS 1.2, que l'on a décrit en première partie. Il apparaît bien que les échanges pré-communication durent 2 allers-retours.

A droite, on peut observer le handshake TLS 1.3, qui ne prend qu'un RTT.

Détaillons les différentes étapes du côté droit afin de comprendre comment fonctionne le handshake TLS 1.3 .

## Etape 1 : Client Hello + Client Key Share

Le client envoie sa Cipher Suite (ensemble des suites cryptographiques proposées) ainsi qu'un "jeu" de paramètres, constitué de groupes de Diffie-Hellman ainsi que de clés prêtes à être échangées en utilisant ces groupes en question. Pour un groupe donné, il envoie plusieurs paires (clé publique (g,p) ; f(clé privée ; clé publique)= $g^a \text{ mod } p$ ) chiffrées par différentes suites cryptographiques (Key Share). Ces clés sont entre autres créées en utilisant un Client.random comme dans TLS 1.2 . L'ensemble des clés générées est contenu dans un champ appelé " Key Share Entry".

Au cours de cette étape, le client fait par ailleurs une hypothèse sur la suite cryptographique que le serveur va probablement choisir. Notons que les clés générées ici sont temporaires, on parle de " ephemeral Diffie-Hellman". Cela va permettre de rajouter encore plus de sécurité par rapport à un Diffie-Hellman classique, qui va permettre de vérifier le principe de Perfect Forward Secrecy (PFS).

Il n'y a a priori pas de demande explicite d'envoi de certificat à cette étape. L'utilisation de certificats est optionnelle et va dépendre de la méthode d'échange de clés sur laquelle s'accordent le client et le serveur. En pratique, dans tous les cas, le client va demander au serveur d'envoyer un certificat X 509, sauf en cas d'utilisation de PSK (Pre Shared Key, cf ce qui suit).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.1.117	172.16.1.130	TCP	74	34152 → 4433 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=269767201 TSecr=0 WS=128
2	0.009143	172.16.1.130	172.16.1.117	TCP	74	4433 → 34152 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=269762153 TSecr=269767201
3	0.010254	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=269767212 TSecr=269762153
4	0.010885	172.16.1.117	172.16.1.130	TLSv1.3	301	Client Hello
5	0.012012	172.16.1.130	172.16.1.117	TCP	66	4433 → 34152 [ACK] Seq=1 Ack=236 Win=30080 Len=0 TSval=269762156 TSecr=269767212
6	0.014010	172.16.1.130	172.16.1.117	TLSv1.3	1139	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
7	0.015756	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=236 Ack=1074 Win=31360 Len=0 TSval=269767217 TSecr=269762158
8	0.017415	172.16.1.117	172.16.1.130	TLSv1.3	146	Change Cipher Spec, Application Data
9	0.017879	172.16.1.117	172.16.1.130	TLSv1.3	124	Application Data, Application Data

▼ TLSv1.3 Record Layer: Handshake Protocol: Client Hello	0000	b8 27 eb 45 99 91 00 0c 29 dd 61 7a 08 00 45 00	..E....).az..E.
Content Type: Handshake (22)	0010	01 1f 85 0b 40 00 40 06 59 b6 ac 10 01 75 ac 10	...@.Y....u...
Version: TLS 1.0 (0x0301)	0020	01 82 85 68 11 51 7f 2a d7 29 ec 60 a9 af 80 18	...h.Q.*).'
Length: 230	0030	00 e5 2a 04 00 00 01 01 08 0a 10 14 52 2c 10 14	...*.....R,..
▼ Handshake Protocol: Client Hello	0040	3e 69 16 03 01 00 e6 01 00 00 e2 03 03 81 47 c1	>i.....G.
Handshake Type: Client Hello (1)	0050	66 d5 1b fa 4b b5 e0 2a e1 a7 87 13 1d 11 aa c6	f...K..*.....
Length: 226	0060	ce fc 7f ab 94 c8 62 ad c8 ab 0c dd cb 20 6f 9d	.....b.....0.
Version: TLS 1.2 (0x0303)	0070	07 f1 95 3e 99 d8 f3 6d 97 ee 19 0b 06 1b f4 84	....>...m.....
Random: 8147c166d51bfa4bb5e02ae1a787131d11aac6cefc7fab94...	0080	0b b6 8f cc de e2 d0 2d 6b 0c 1f 52 53 13 00 08	.....-k...RS...
Session ID Length: 32	0090	13 02 13 03 13 01 00 ff 01 00 00 91 00 00 00 0c	.....dogfish...
Session ID: 6f9d07f1953e99d8f36d97ee190b061bf4840bb68fccdee2...	00a0	00 0a 00 00 07 64 6f 67 66 69 73 68 00 0b 00 04	.....
Cipher Suites Length: 8	00b0	03 00 01 02 00 0a 00 0c 00 0a 00 1d 00 17 00 1e	.....
▼ Cipher Suites (4 suites)			

## Capture Client Hello TLS 1.3 (Cloudshark)

On remarque entre autres la présence du Session\_ID et d'une liste de 4 Cipher Suites proposées (parmi les 5 au total acceptées dans TLS 1.3).

On peut faire une autre remarque : le fait que l'identifiant de session Session\_ID soit non nul lors du Client Hello provient du fait que notre capture est réalisée à l'aide de l'outil en ligne Cloud Shark, qui se comporte comme une " Middlebox" (réseau qui inspecte, filtre, manipule

des paquets). La RFC de TLS 1.3 (<https://tools.ietf.org/html/rfc8446#appendix-D.4>) explique que pour les Middlebox, certains comportements particuliers sont observés, comme la nécessité d'envoyer un identifiant de session non nul même lors du Client Hello. Pour un handshake classique, cet identifiant de session est nul lors du Client Hello, et c'est le serveur qui le crée lors du Server Hello.

#### ▼ Cipher Suites (4 suites)

```
Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)
```

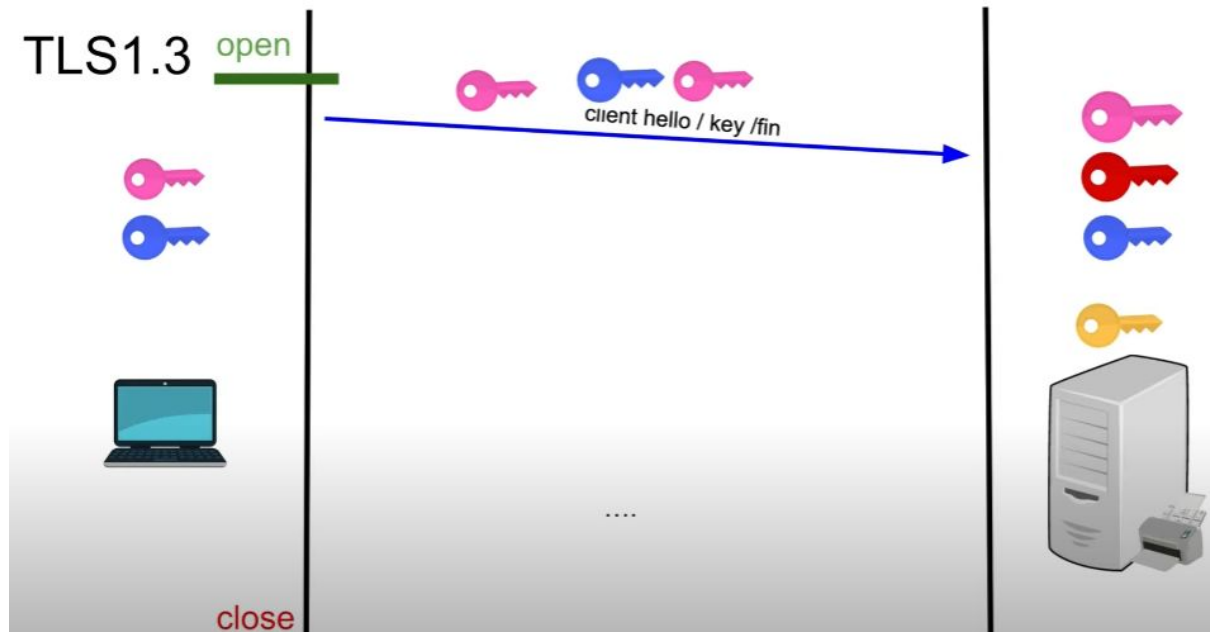
On remarque également une liste d'extensions possibles, entre autres le Key Share :

```
Extensions Length: 145
▶ Extension: server_name (len=12)
▶ Extension: ec_point_formats (len=4)
▶ Extension: supported_groups (len=12)
▶ Extension: SessionTicket TLS (len=0)
▶ Extension: encrypt_then_mac (len=0)
▶ Extension: extended_master_secret (len=0)
▶ Extension: signature_algorithms (len=30)
▶ Extension: supported_versions (len=7)
▶ Extension: psk_key_exchange_modes (len=2)
▶ Extension: key_share (len=38)
```

#### **Etape 2 : Server Hello + Server Key Share (+ Certificate optionnel + Certificate Request optionnel + Certificate Verify optionnel + EncryptedExtensions+ Hello Retry en cas d'échec) + Server Finished**

Le serveur reçoit la Shared Key du client. Il peut ensuite construire sa propre clé (en l'occurrence paire de clés avec Diffie-Hellman) avec la primitive cryptographique qu'il a retenues parmi celles proposées par le client (s'il en supporte au moins une). Cette clé (Key Share) doit être créée en utilisant la même suite cryptographique que le client, et si DH "ephemeral" a été utilisé par le client, le serveur doit également utiliser le même groupe de Diffie-Hellman pour partager  $f(\text{clé privée serveur} = b ; \text{clé publique} = (g,p)) = g^b \text{ mod } p$ . Si jamais aucune des primitives proposées par le client n'est supportée par le serveur, ce dernier envoie un message "Hello Retry" pour que le client fasse un nouveau Client Hello avec une autre primitive cryptographique qu'il supporte. Dans le cas où le client et le serveur ne partageraient aucune primitive cryptographique (cas rare), le serveur envoie un message Alert et la communication est stoppée.

Notons qu'à ce moment-là, le serveur dispose alors des deux Key Share (client et serveur), il peut donc construire la clé symétrique finale, le master secret.



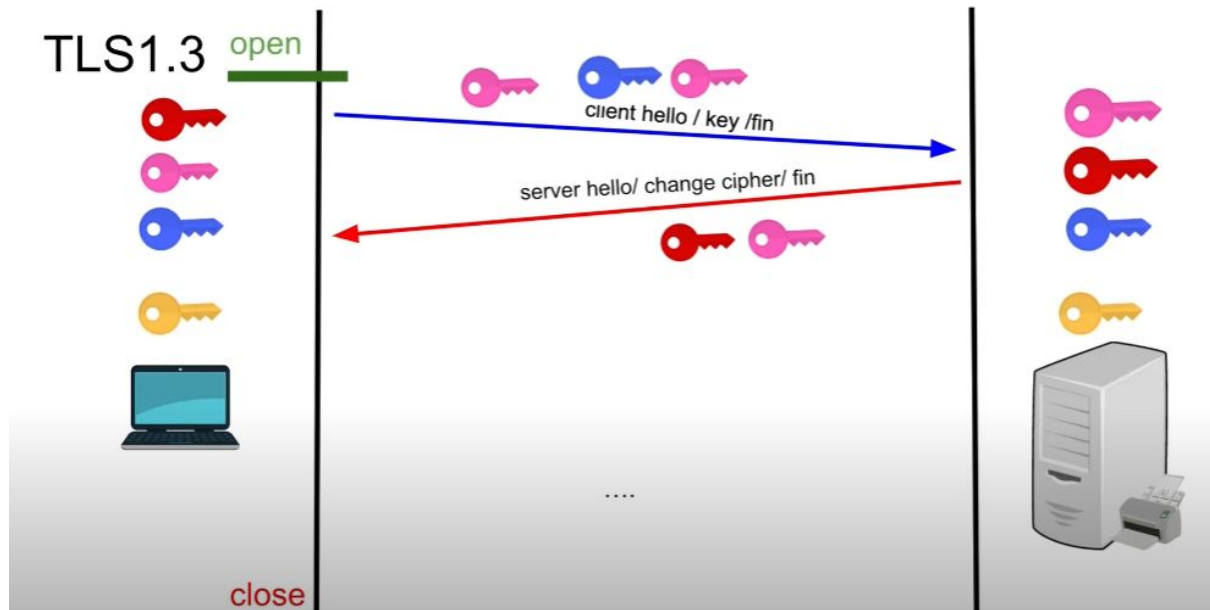
-  Clé privée serveur
-  Clé publique client
-  Clé privée client
-  Clé symétrique finale

Sur ce schéma, on peut observer la paire mélangée de clés publique/privée du client. La clé rose est la clé publique, la clé bleue est sa clé privée. Ce qu'il envoie en clair est la paire constituée de sa clé publique rose, ainsi que le mélange clé publique/privée (clé rose + clé bleue). A partir de cette paire, il est impossible de remonter à la clé privée bleue (cela reviendrait à résoudre un problème de Diffie-Hellman, mathématiquement complexe) pour un attaquant.

Le serveur, de son côté, génère une clé privée temporaire, en rouge sur ce schéma. En la combinant à la paire clé bleue/clé rose envoyée par le client, il peut construire la clé symétrique finale (ou bien du moins qui sera donnée en entrée à la suite cryptographique choisie pour construire la clé finale) ! C'est la clé jaune sur le schéma.

Le serveur envoie alors au client le mélange clé rouge (sa clé privée) / clé publique du client. Encore une fois, à partir de ce mélange de clés, on ne peut retrouver la clé privée du serveur. Notons par ailleurs quand dans le mode Diffie-Hellman, le serveur prouve qu'il est

bien propriétaire de sa clé privée en utilisant une signature numérique permettant en plus de vérifier l'intégrité de l'échange: cela empêche l'attaque connue man in the middle sur Diffie-Hellman. Il signe en effet tout le handshake avec sa clé privée dans le certificat Verify. Voici alors la situation :



En combinant la paire clé rouge/clé rose envoyée par le serveur ainsi que sa clé privée bleue, le client peut construire une clé partagée. C'est la même que celle créée par le serveur ! Cela est dû à l'associativité des opérations mises en jeu par Diffie-Hellman. (ainsi que des suites cryptographiques). Pour obtenir la clé symétrique finale qui va être utilisée pour le cryptage des messages, de chaque côté, on mélange le secret partagé (PMS) avec le transcript-hash de l'échange de clé, cela donne un HMS. A partir du HMS et des randoms envoyés, on produit un tk\_hs qui va être la clé symétrique finale. Ce contexte changeant à chaque fois par le biais des ClientHello.random et ServerHello.random permet de s'assurer que la clé symétrique finale partagée est bien différente à chaque handshake.

Ainsi, on observe que la clé symétrique partagée est bien créée au niveau client et niveau serveur, et ce en un seul aller-retour ! L'utilisation de Diffie-Hellman ("ephemeral") a donc un apport double : un gain en terme de sécurité (PFS) mais également en terme de nombre d'échanges, puisque la construction de la clé symétrique finale a été réalisée en un seul aller-retour ! C'est le grand apport de TLS 1.3 .

Le serveur doit par ailleurs envoyer les Encrypted Extensions, qui constituent le premier message qui va être crypté à l'aide de clés dérivées du PMS. Le client doit absolument vérifier les Encrypted Extensions afin de s'assurer qu'aucune des extensions utilisées n'est interdite, auquel cas un message Alert "illegal\_parameter" sera lancé. Les Encrypted Extensions peuvent être par exemple le Server Name, le Message Length...

Le serveur peut lors de cette étape envoyer son certificat ("Certificate") si la méthode d'échange choisie le nécessite. Notons que ce certificat est déjà crypté puisque la clé symétrique finale est construite par le serveur à ce niveau. A la réception du Server Hello, le



client va également pouvoir construire cette clé symétrique et donc décoder le certificat du serveur. également lui même demander un certificat d'authentification au client s'il estime que cela est nécessaire ("Certificate Request").

Le serveur envoie aussi un Certificate Verify qui permet de prouver la possession de la clé privée. Ce certificate verify est obligatoire si le serveur a envoyé un certificat. Ce champ permet aussi de s'assurer de l'intégrité du handshake jusqu'à ce point. En effet, le serveur signe avec sa clé privée le transcript-hash qui est le haché des messages du handshake concaténés.

Le serveur envoie alors un "Server Finished". Ce finished est un MAC sur tout le handshake. Il permet de montrer la bonne possession du PMS  $g^{(ab)}$ . En effet, ce Server finished est calculé de la manière suivante :  $\text{ServerFinished} = \text{HMAC}(\text{finished\_key},$

$\text{Transcript-Hash}(\text{Handshake Context},$   
 $\text{Certificate}^*, \text{CertificateVerify}^*))$

$\text{finished\_key}$  est dérivée du PMS ( $g^{(ab)}$ ). Cela permet de lier les identités des 2 points.

No.	Time	Source	Destination	Protocol	Length	Info
2	0.009143	172.16.1.130	172.16.1.117	TCP	74	4433 → 34152 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=269762153 TSecr=269762153
3	0.010254	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=269767212 TSecr=269762153
4	0.010885	172.16.1.117	172.16.1.130	TLSv1.3	301	Client Hello
5	0.012012	172.16.1.130	172.16.1.117	TCP	66	4433 → 34152 [ACK] Seq=1 Ack=236 Win=30080 Len=0 TSval=269762156 TSecr=269767212
6	0.014010	172.16.1.130	172.16.1.117	TLSv1.3	1139	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data
7	0.015756	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=236 Ack=1074 Win=31360 Len=0 TSval=269767217 TSecr=269762158
8	0.017415	172.16.1.117	172.16.1.130	TLSv1.3	146	Change Cipher Spec, Application Data
9	0.017879	172.16.1.117	172.16.1.130	TLSv1.3	124	Application Data, Application Data
10	0.024132	172.16.1.130	172.16.1.117	TLSv1.3	321	Application Data

Handshake Protocol: Server Hello Handshake Type: Server Hello (2) Length: 118 Version: TLS 1.2 (0x0303) Random: 3964dbec5022bfb0d783a15f8fd02518c8cf05b901c389b... Session ID Length: 32 Session ID: 6f9d07f1953e99d8f36d97ee190b061bf4840bb68fccdee2... Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302) Compression Method: null (0) Extensions Length: 46 Extension: supported_versions (len=2) Extension: key_share (len=36)	0000 b8 27 eb 45 99 91 00 0c 29 ee 6c 65 08 00 45 00 ..E....).le..E. 0010 04 65 97 c5 40 00 40 06 43 b6 ac 10 01 82 ac 10 .e..@.C..... 0020 01 75 11 51 85 68 ec 60 a9 af 7f 2a d8 14 80 18 ..u.Q.h'....*... 0030 00 eb 48 48 00 00 01 01 08 0a 10 14 3e 6e 10 14 ..HH.....>n... 0040 52 2c 16 03 03 00 7a 02 00 00 76 03 03 39 64 db R,....z...v..9d. 0050 ec 50 22 bf bd 07 83 a1 5f 8f d0 25 18 c8 cf 05 ..P".....%.... 0060 b6 90 1c 38 9b 8a 28 46 39 e3 7c db 66 20 6f 9d ...8..(F9.).f o.. 0070 07 f1 95 3e 99 d8 f3 6d 97 ee 19 0b 06 1b f4 84 ...>...m..... 0080 0b b6 8f cc de e2 d0 2d 6b 0c 1f 52 53 13 13 02 .....-k..RS... 0090 00 00 2e 00 2b 00 02 7f 1c 00 33 00 24 00 1d 00 .....+.....3.\$... 00a0 20 ee d2 11 97 9a c7 94 1f dd de 11 1c d4 f5 b9 ..... 00b0 81 ec d0 69 bb f9 e3 ef f2 b4 2d 01 cb 6b 9e 57 ...i......k.W
--	---

### Capture Server Hello (Cloudshark)

On peut notamment observer la Cipher Suite retenue par le serveur (TLS\_AES\_256\_GCM\_SHA384) qui fait bien partie des 4 Cipher Suites proposées par le client dans le Client Hello.

On remarque également un identifiant de session Session\_ID identique à celui du Client Hello, ce qui confirme que les 2 ont bien une session commune.

### **Etape 3 : Client Finished ( + Certificate optionnel + Certificate Verify optionnel) + Application Data**

A ce niveau, si le serveur a accepté une des primitives cryptographiques proposées, le client a pu construire la clé symétrique partagée à l'aide du processus expliqué ci-dessus. Il peut avec cette clé décoder le certificat du serveur (s'il y en a besoin). Le client peut également envoyer son propre certificat, chiffré avec la clé secrète partagée (master secret PMS) si le serveur a fait une Certificate Request.

Si le client envoie un certificat, il doit aussi envoyer un Certificate Verify à l'image de celui du serveur: une signature par sa clé privée du contexte.

Le client envoie finalement un Client Finished à l'image du Server Finished.

A partir de ce moment là, le transfert de données applicatives (Application Data) peut débuter.

Notons que si le serveur n'a retenu aucune des primitives proposées par le client, et donc envoyé un Hello Retry, cette étape sera un nouveau Client Hello avec une nouvelle proposition de CipherSuite par le client. Si aucun accord n'est trouvé même suite à ce nouveau Client Hello, la communication est stoppée suite à un message Alert du serveur.

The image shows a Wireshark packet capture of a TLS 1.2 handshake and subsequent data exchange. The packet list on the left shows packets 5 through 13. Packet 6 is selected, showing details for the TLSv1.3 Record Layer, Change Cipher Spec Protocol, and Application Data. The packet bytes pane on the right shows the raw data in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Length	Info
5	0.012012	172.16.1.130	172.16.1.117	TCP	66	4433 → 34152 [ACK] Seq=1 Ack=236 Win=30080 Len=0 TSval=269762156 TSecr=269767212
6	0.014010	172.16.1.130	172.16.1.117	TLSv1.3	1139	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
7	0.015756	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=236 Ack=1074 Win=31360 Len=0 TSval=269767217 TSecr=269762158
8	0.017415	172.16.1.117	172.16.1.130	TLSv1.3	146	Change Cipher Spec, Application Data
9	0.017879	172.16.1.117	172.16.1.130	TLSv1.3	124	Application Data, Application Data
10	0.024132	172.16.1.130	172.16.1.117	TLSv1.3	321	Application Data
11	0.024250	172.16.1.130	172.16.1.117	TLSv1.3	321	Application Data
12	0.024500	172.16.1.130	172.16.1.117	TCP	66	4433 → 34152 [FIN, ACK] Seq=1584 Ack=375 Win=30080 Len=0 TSval=269762168 TSecr=269767219
13	0.025876	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=375 Ack=1585 Win=35712 Len=0 TSval=269767227 TSecr=269762168

Transmission Control Protocol, Src Port: 34152, Dst Port: 4433, Seq: 236, Ack: 1074

Secure Sockets Layer

- TLSv1.3 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
  - Content Type: Change Cipher Spec (20)
  - Version: TLS 1.2 (0x0303)
  - Length: 1
  - Change Cipher Spec Message
- TLSv1.3 Record Layer: Application Data Protocol: Application Data
  - Opaque Type: Application Data (23)
  - Version: TLS 1.2 (0x0303)
  - Length: 69
  - Encrypted Application Data: 3e52f513cc90c0b7064d43c6543b53a547397d8b93ce

0000 b8 27 eb 45 99 91 00 0c 29 dd 61 7a 08 00 45 00 .....E.....az..E.  
0010 00 84 85 0d 40 00 40 06 5a 4f ac 10 01 75 ac 10 ...@.ZD...u..  
0020 01 82 85 68 11 51 7f 2a d8 14 ec 60 ad e0 80 18 ...h.Q\*.....  
0030 00 f5 73 d6 00 00 01 01 08 0a 10 14 52 33 10 14 ...S.....R3..  
0040 3e 6e 14 03 03 00 01 01 17 03 03 00 45 3e 52 f5 >n.....ER.  
0050 13 cc 90 0c 0b 70 64 d4 3c 65 43 b5 3a 54 73 97 ....pd.<eC.:Ts.  
0060 d8 b9 3c e9 fb 72 3a ca 83 e1 dd a1 ba 45 f4 be ...<.r.....E..  
0070 1f a6 94 77 6a 08 21 1c c9 5b 11 fd f7 b1 fc a5 ...Wj!..[.....  
0080 f4 05 47 df f5 3f de 0f 22 db 57 21 b8 f9 73 be ..G...".Wl..s.  
0090 3f db ?.

Capture Change Cipher Spec (TLS 1.2 + Application Data ) (Cloudshare)

Cette capture montre le deuxième RTT lors d'un échange avec TLS 1.3 . On remarque la présence d'Application Data dès le deuxième aller-retour, ce qui est caractéristique de TLS 1.3 . Fait plus étrange : on remarque l'existence d'un champ Change Cipher Spec, qui est obsolète depuis TLS 1.3 . Cela est encore dû à la remarque faite plus haut, concernant le fait que les “ Middlebox “ adoptent un comportement particulier : pour citer la RFC “ Implementations can increase the chance of making connections through those middleboxes by making the TLS 1.3 handshake look more like a TLS 1.2 handshake.” C'est pour cette raison que l'on observe le champ Change Cipher Spec, qui est en fait un “ dummy Change Cipher Spec “, c'est-à-dire un CCS “camouflé” pour donner l'impression au serveur d'être en TLS 1.2 et maximiser les chances d'établissement de connexion avec le client.

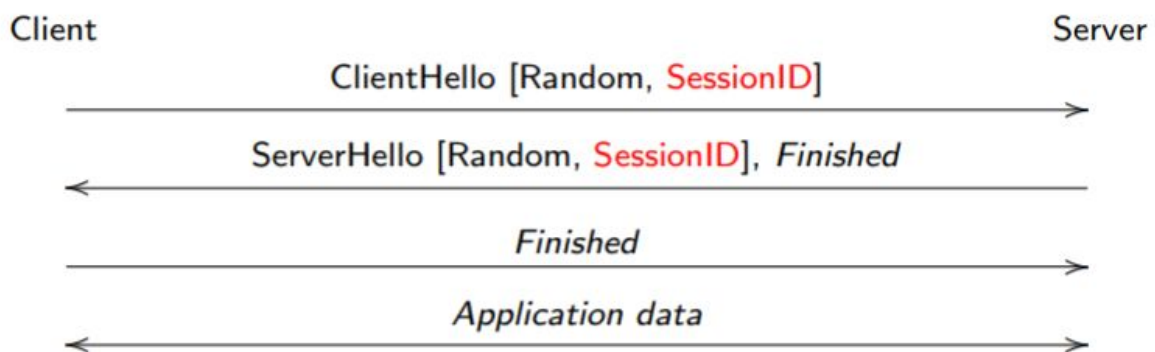
### c) Reprise d'une connexion

#### TLS 1.2

Pour TLS 1.2, la reprise de connexion était gérée de 2 manières possibles :

- soit par l'utilisation de l'identifiant de session Session\_ID, gardé en mémoire cache du serveur. L'échange de reprise durait 1 RTT, au cours duquel le client envoyait un nouveau Client.random ainsi que le Session\_ID, tandis que le serveur vérifiait dans

son cache qu'ils avait bien une session commune avec le Session\_ID en question, tout en envoyant également à son tour un nouveau Server.random. Cela permettait de construire une nouvelle clé symétrique partagée, dérivée de ces deux Randoms ainsi que de l'ancien master secret. Cette possibilité présentait un risque important dans la mesure où la compromission du master secret ou de la clé privée du serveur entre temps permettrait à un attaquant d'avoir accès au nouveau master secret. Notons qu'il est préconisé que l'identifiant de session reste en mémoire cache du serveur pendant 24h, sans quoi le client et le serveur devront à nouveau effectuer un full handshake.



- Une autre possibilité, plus facile à mettre en oeuvre en terme de mémoire pour le serveur, est l'utilisation d'un ticket de session. En effet, dans l'implémentation avec le Session\_ID, le serveur a la responsabilité de stocker les Session\_ID de tous les clients avec lesquels il a communiqué pendant une durée donnée. Cette pose des problèmes de scalabilité pour le serveur dans la mesure où Internet comporte un très grand nombre d'utilisateurs par seconde. L'idée de l'utilisation d'un ticket est que c'est le client qui va devoir garder en mémoire de quoi pouvoir se connecter à nouveau au serveur. A la fin d'un handshake TLS, le serveur envoie un ticket au client ainsi que des informations clés sur la session en cours, chiffrés avec sa clé privée. Le client va garder en mémoire cache le ticket et les informations associées, et lorsqu'il voudra reprendre une session avec le serveur, il lui enverra ces informations que seul le serveur pourra décoder. La session pourra alors reprendre

### TLS 1.3 :

Sous TLS 1.3, il faut savoir que la reprise de session via des tickets de session ou bien un identifiant de session est obsolète. Ces deux méthodes sont remplacées par ce que l'on appelle un mode PSK (Pre-Shared-Key). Ce PSK existait déjà dans TLS 1.2 mais n'était pas utilisé dans ce contexte, seulement pour certaines applications de type IoT.

A la fin du handshake d'une session précédente, le PSK est échangé entre le client et le serveur afin d'être utilisé par le client lors d'une nouvelle connexion. Ce PSK est construit par le serveur à partir d'une clé dérivée du master secret, puis envoyé au client, qui le garde en mémoire cache. Il contient également des informations sur la configuration du serveur. Le client peut alors utiliser par la suite cet identifiant PSK qui, s'il est accepté par le serveur, va



permettre une connexion sécurisée à partir de paramètres cryptographiques utilisés lors de la dernière session. La clé dérivée du master secret va être utilisée pour chiffrer les nouvelles communications.

Le client envoie une ou plusieurs identités PSK sous forme d'un ensemble de données opaques. Il peut s'agir de clés de recherche de base de données (similaires aux identificateurs de session) ou de valeurs auto-chiffrées et auto-authentifiées (similaires aux tickets de session). Si le serveur accepte l'une des identités PSK données, il répond avec celle qu'il a sélectionnée. L'extension KeyShare est envoyée pour permettre aux serveurs d'ignorer les PSK et de revenir à un handshake complet.

Par ailleurs, le PSK peut être combiné à une clé d'échange de type Diffie-Hellman, ce qui permet de vérifier la propriété de Perfect Forward Secrecy, qui n'est pas garantie en cas d'utilisation du PSK seul.

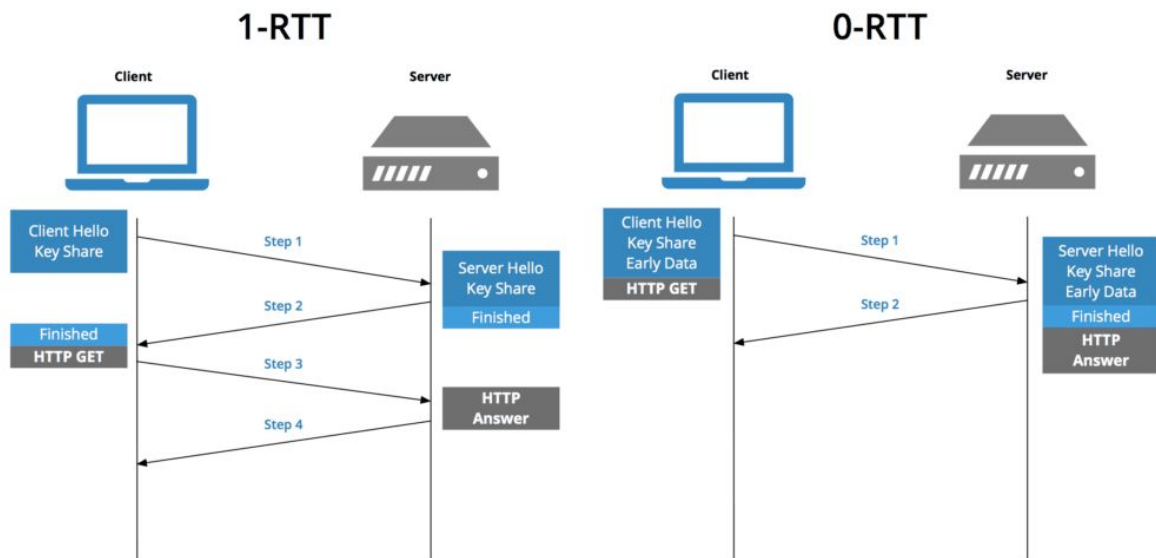
Le handshake se déroule selon le schéma classique d'initiation de session, à ceci près que :

- Le client envoie une Pre Shared Key identity en précisant le mode d'envoi (PSK seul ou PSK+keyshare)
- Le serveur envoie la Pre Shared Key Identity choisie
- Aucun certificat n'est mis en jeu (ni Certificate, Certificate Request, Certificate Verify)

#### **d) Option “ 0-RTT “**

Le 0-RTT est une des options phares de TLS 1.3 . Il permet qu'un client et un serveur s'envoient des données applicatives dès le premier aller-retour lors d'une reprise de session, s'ils ont déjà communiqué auparavant. C'est une option inspirée du protocole QUIC proposé par Google. Elle est basée sur l'utilisation du PSK présentée dans la partie b). La reprise de connexion va se dérouler en mode 0-RTT si, lors du Client Hello, le client envoie l'extension “ early\_data “. Dans ce cas, il pourra envoyer directement des données applicatives, auxquelles le serveur répondra avec succès si son PSK est accepté.

Cela représente un gain d'un aller-retour par rapport au mode de reprise classique, également présent dans TLS 1.2 .



On peut toutefois remarquer que le 0-RTT est en deçà des autres mesures présentées par TLS 1.3 en termes de sécurité. En effet :

- la PFS n'est pas vérifiée, étant donné que les premières données applicatives échangées sont chiffrées avec la clé dérivée contenue dans le PSK et fournie par le serveur lors de la connexion précédente. Comme celle-ci se base sur la clé privée du serveur, une compromission de celle-ci permettrait à un attaquant de décrypter les messages envoyés.
- il n'y a pas de garantie de protection contre une attaque "replay". En effet, sous la reprise de session classique de type 1-RTT, le Server random permettait de servir de nonce empêchant le replay.

### 3) Gestion des services cryptographiques

#### a) Gestion de l'authentification

##### TLS 1.2 :

Dans le cas de l'authentification du serveur : le client utilise la clé publique que le serveur lui a donnée lors du "server hello" afin de chiffrer les données qui seront (notamment le pre master secret) qui seront utilisées pour générer la clé secrète. Ainsi, le serveur peut générer la clé secrète avec le client si et seulement s'il peut décoder ces données avec sa clé privée. (qu'il est le seul à avoir).

Dans le cas de l'authentification du client : le serveur utilise la clé publique fournie dans le certificat du client afin de décoder les données que ce dernier lui a envoyées durant l'étape 5. Le principe est le même, l'authentification est alors assurée.

Les échanges des messages "finished" avec la clé partagée permettent de s'assurer que l'authentification est complète. De plus, l'authentification est complète si et seulement si chacune des étapes du handshake s'est déroulée correctement.

Le rôle des certificats permet aussi de garantir d'autant plus l'authentification. Leur gestion est assez complexe, mais ce qu'il faut retenir est que les certificats SSL jouent un rôle très similaire aux certificats client. Toutefois, dans le cas d'un client, le certificat est utilisé pour identifier le client/individu lui-même tandis que dans le cas d'un serveur le certificat authentifie le propriétaire du site.

Si on appelle CA Client l'autorité de certification du Client et CA Serveur l'autorité de certification du Serveur, dans le cas d'une authentification client et serveur, le serveur a besoin :

- d'un certificat personnel fourni par CA Serveur
- de sa clé privée
- d'un certificat du client fourni par CA Client

Le client quant à lui a besoin :

- d'un certificat personnel fourni par CA Client
- de sa clé privée
- d'un certificat du serveur fourni par CA Server

On observe qu'en termes d'authentification, les échanges se font à chaque fois de manière similaire pour le client et le serveur.

Les certificats contiennent diverses informations comme un numéro de série, une date de validité, des données personnelles sur le propriétaire (nom, adresse, e-mail...)

### **TLS 1.3 :**

Le serveur doit toujours s'authentifier, pour le client cela est optionnel. L'authentification peut avoir lieu dans le cas du handshake TLS 1.3 en utilisant de la cryptographie asymétrique.

Le serveur peut s'authentifier via un certificat x509 dans l'initialisation de la session.

Le Certificate Verify permet de s'assurer de la possession par le serveur de la clé privée qu'il prétend avoir.

Dans le cas d'une reprise de session, le client envoie le PSK identity ainsi qu'un PSK binder (haché du contexte et d'une clé dérivée du PSK) permettant de prouver au serveur qu'il possède bien le PSK.

Le serveur, quant à lui, permet lui aussi de prouver au client qu'il possède bien le PSK par le biais du Server Finished qui est toujours obligatoire et qui fait partie du bloc

d'authentification. En effet, dans le cas d'une reprise de session, le Server Finished est calculé par  $H(\text{base\_key}, \text{transcript-hash})$ . La base\_key est dérivée du PSK et du contexte. Ainsi, le client peut vérifier le haché pour s'assurer que le serveur a bien le PSK.

### **b) Vérification des certificats**

Il existe 4 étapes lors des vérifications de certificats :

- la vérification de la signature digitale
- la "chaîne de certificats" est vérifiée : normalement le client et le serveur ont dû avoir des certificats intermédiaires
- la date d'expiration, d'activation et la période de validité sont vérifiées
- l'état de révocation du certification

### **c) Gestion de la confidentialité**

D'une part, on observe que l'échange est basé sur une alternance de chiffrements symétriques et asymétriques, ce qui renforce la sûreté de la communication.

Afin de minimiser le risque en cas de vol ou d'altération de la clé secrète, cette dernière peut être renégociée périodiquement, et l'ancienne n'est alors plus valide (Randoms à intervalles réguliers pour TLS 1.2 et 1.3 + "ephemeral Diffie-Hellman" pour TLS 1.3). Le même principe de sûreté est utilisée lors de l'envoi de chaînes aléatoires comme expliquée ci-dessus, qui permettent d'éviter les attaques "man in the middle".

Durant le handshake, le client et le serveur s'accordent sur la primitive cryptographique utilisée ainsi que la clé secrète qui sera donc utilisée seulement pour la session en cours. Tous les messages au cours de la session seront donc chiffrés à la fois avec la primitive choisie et la clé secrète partagée, ce qui permet de s'assurer que le message reste privé et ne sera pas intercepté. Les chaînes aléatoires ne font que diminuer encore plus le risque déjà faible d'interception.

Par ailleurs, pour TLS 1.2, le fait que le chiffrement utilisé pour transmettre la clé privée soit asymétrique évite tout problème de distribution de la clé (pas de risque d'interception).

Pour TLS 1.3, l'utilisation de Diffie-Hellman garantit la confidentialité, et même la Perfect Forward Secrecy, dans la mesure où les clés transférées en clair (paire clé publique / mélange clé publique + clé privée) ne permettent pas de remonter à la clé privée pour un attaquant, car elles nécessiteraient de résoudre un problème mathématique computationnellement complexe.

### **d) Gestion de l'intégrité**

L'intégrité est principalement permise par le calcul de condensats (hash) des messages avec clé secrète (hmac). Les fonctions de hachage utilisées sont fournies dans la

liste des suites cryptographiques utilisées par le client, et c'est ensuite le serveur qui va faire le choix final, qu'il communiquera au client. La fonction de hachage utilisée est donc clairement définie de part et d'autre. Pour ceci, il est toutefois important que la liste d'algorithmes cryptographiques supportés par le client ne soit pas de type "None", autrement l'intégrité est fortement compromise.

Dans TLS 1.3, l'intégrité du handshake est aussi assurée par les HMAC du transcript-hash, que ce soit dans le certificate verify, ou bien dans le Finished.

### **e) Gestion du non-rejeu dans TLS 1.3**

Pour le 1-RTT, le non-rejeu est assuré par les numéros de séquence contenus dans les paquets au cours d'une même connexion.

Pour le 0-RTT cependant, il est possible d'effectuer une attaque "replay".

## **II Benchmarking**

### **1) Introduction**

Le but de cette partie est de mesurer les performances de TLS 1.3 à échelle moyenne. Pour cela, nous avons besoin de mettre en place un environnement virtuel de simulation. On utilise 2 machines virtuelles Kali toutes les deux en mode bridge sur un réseau LAN privé. La première machine Kali joue le rôle de(s) client(s) et la deuxième de serveur. Le but étant de simuler un trafic HTTP/2 basé sur du TLS1.3 avec plusieurs clients et plusieurs requêtes par client.

Pour mesurer les performances, on peut regarder plusieurs paramètres: le temps de connexion, le nombre de requêtes traitées par seconde, le temps de réponse à une requête, et le TTFB (Time to first byte), qui correspond à la durée entre le moment où le client envoie sa requête HTTP et celui où il reçoit le premier octet de la réponse serveur.

Ce TTFB est composé du temps de connexion, du temps d'envoi de la requête HTTP, et de celui de réception du premier octet de réponse. On cherche à avoir ce TTFB le plus faible possible car il est indicateur d'une application serveur bien configurée.

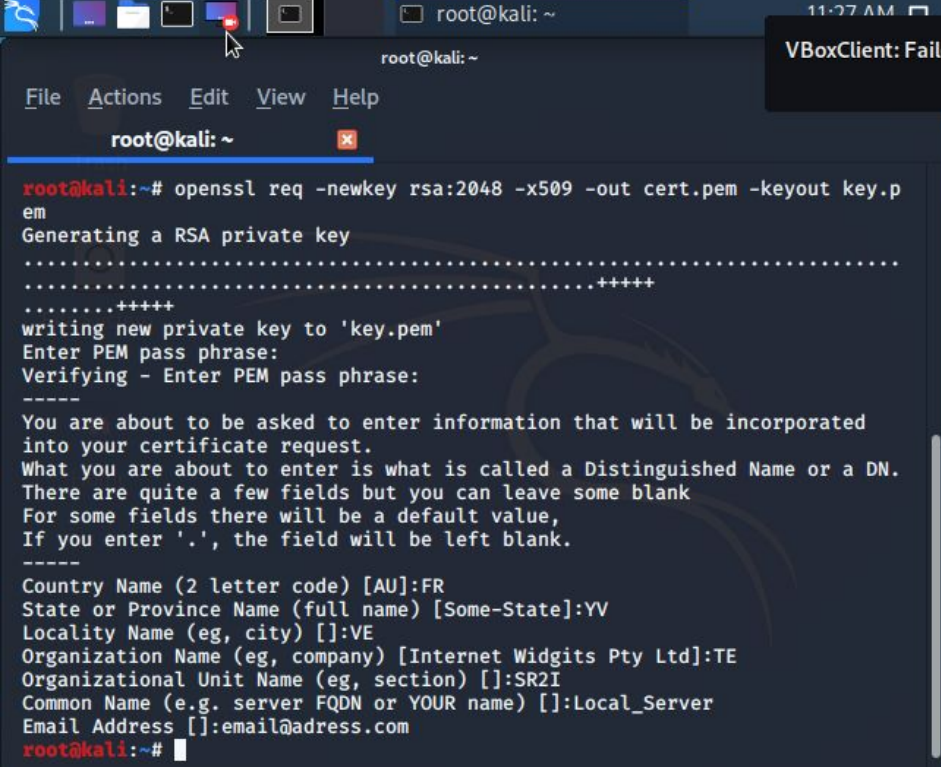
Si on veut se concentrer sur les performances de TLS 1.3 uniquement, on peut se concentrer uniquement sur le temps de connexion et les performances des algorithmes de cryptage utilisés.

### **2) Temps de connexion:**

Commençons par étudier le temps de connexion dans le cas simple d'un seul client.

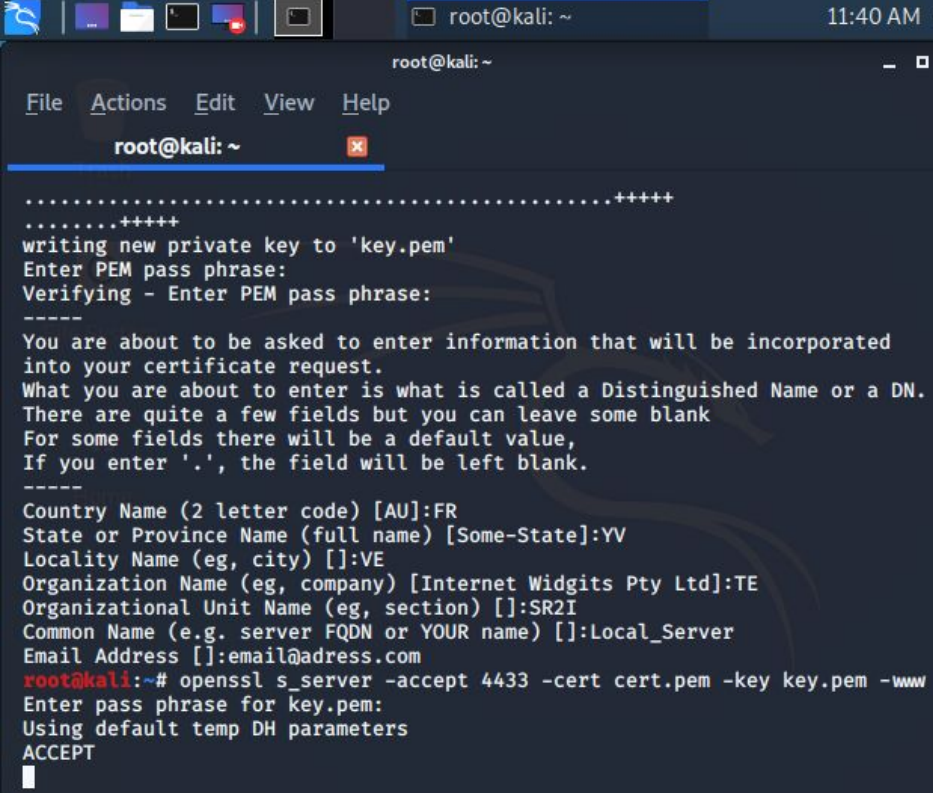
Nous devons tout d'abord mettre en place un serveur HTTP qui supporte TLS 1.2 et TLS 1.3. Pour cela nous utilisons openssl.

Tout d'abord, nous devons générer un certificat auto-signé et une paire de clé aux formats pem. Pour cela, nous utilisons la commande:



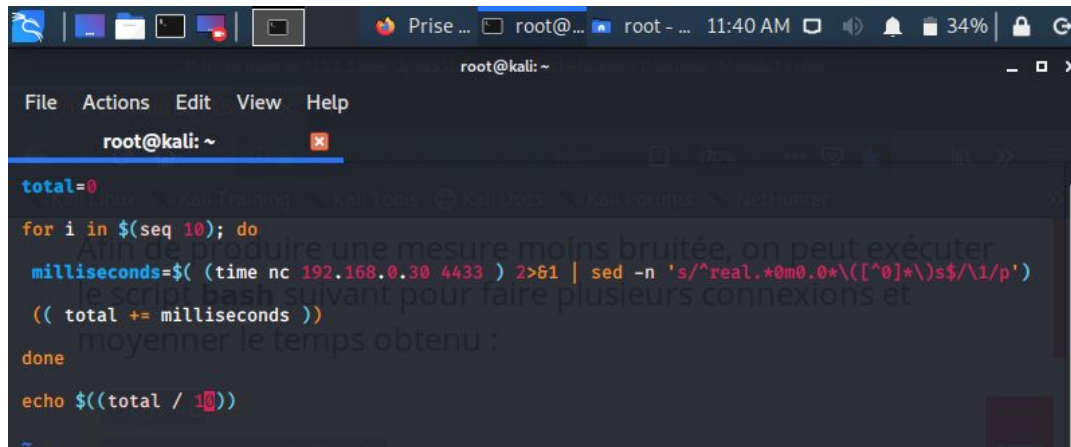
```
root@kali: ~
File Actions Edit View Help
root@kali: ~
root@kali:~# openssl req -newkey rsa:2048 -x509 -out cert.pem -keyout key.p
em
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:YV
Locality Name (eg, city) []:VE
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TE
Organizational Unit Name (eg, section) []:SR2I
Common Name (e.g. server FQDN or YOUR name) []:Local_Server
Email Address []:email@adress.com
root@kali:~#
```

Ceci fait, nous pouvons lancer le serveur https sur le port 4433 grâce à la commande:



```
root@kali: ~
File Actions Edit View Help
root@kali: ~
.....+++++
.....+++++
writing new private key to 'key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:YV
Locality Name (eg, city) []:VE
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TE
Organizational Unit Name (eg, section) []:SR2I
Common Name (e.g. server FQDN or YOUR name) []:Local_Server
Email Address []:email@adress.com
root@kali:~# openssl s_server -accept 4433 -cert cert.pem -key key.pem -www
Enter pass phrase for key.pem:
Using default temp DH parameters
ACCEPT
█
```

Nous voulons d'abord déterminer la valeur du RTT entre le client et le serveur. Pour cela, nous écrivons le script suivant:



```
total=0
for i in $(seq 10); do
  milliseconds=$((time nc 192.168.0.30 4433 ) 2>&1 | sed -n 's/^real.*m0.0*\\([^\0]*\\)s$/\1/p'))
  (( total += milliseconds ))
done
echo $((total / 10))
```

Ce script permet en utilisant la commande time de mesurer le RTT entre le client et le serveur grâce à une moyenne sur 10 essais.

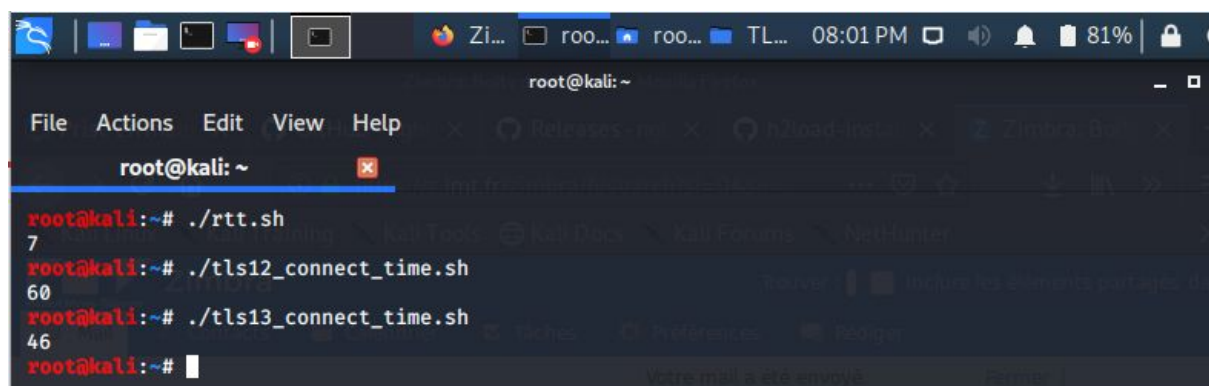
Ceci fait, on écrit des scripts similaire en remplaçant la commande time nc par la commande:

```
time openssl s_client -tls1_2 -connect {Adresse Serveur}:4433
```

ou bien

```
time openssl s_client -tls1_3 -connect {Adresse Serveur}:4433
```

Les résultats sont :



```
root@kali:~# ./rtt.sh
7
root@kali:~# ./tls12_connect_time.sh
60
root@kali:~# ./tls13_connect_time.sh
46
root@kali:~#
```

Nous voyons que la connection via TLS 1.3 prend près de 14 ms de moins que TLS 1.2. En théorie, la différence ne devrait être que d'un RTT soit de 7 ms. Cependant, il faut aussi prendre en compte le temps que met le processus côté serveur (et client !) à traiter chaque requête. Plus le serveur reçoit de paquets côté client, plus il a de traitement à faire. Il faut aussi compter le bruit de la mesure ainsi que la congestion réseau.

On a donc un temps de connection de  
60ms(TLS1.2)=46ms(TLS1.3)+7ms(1RTT)+7ms(temps de traitement de 2 requêtes  
supplémentaires+congestion+bruit\_mesure)

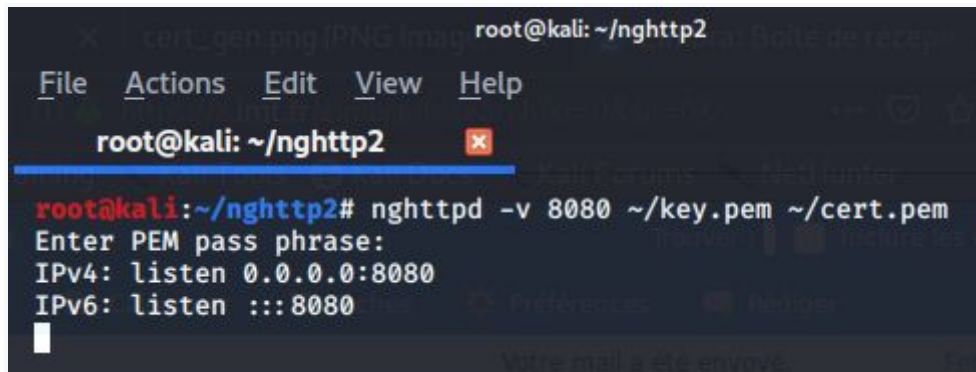
### 3) Modélisation de plusieurs clients

La mesure du temps de connection pour un client est certe importante. Cependant, le protocole est fait pour être déployé à grande échelle. Il faut donc mesurer les performances de ce protocole dans le cadre de plusieurs clients effectuant chacun plusieurs requêtes sur le même serveur.

L'outil que nous allons utiliser est h2load (<https://github.com/nghttp2/nghttp2>). Le serveur openssl ouvert tout à l'heure ne supportant pas HTTP /2, nous décidons d'utiliser le même module, nghttp2, par soucis de compatibilité.

Ainsi, on démarre un serveur grâce à la commande:

```
$ nghttpd -v 8080 ~/key.pem ~/cert.pem
```

A terminal window titled 'root@kali: ~/nghttp2' showing the command 'nghttpd -v 8080 ~/key.pem ~/cert.pem' being executed. The output shows 'Enter PEM pass phrase:', 'IPv4: listen 0.0.0.0:8080', and 'IPv6: listen :::8080'.

```
root@kali: ~/nghttp2
File Actions Edit View Help
root@kali: ~/nghttp2
root@kali:~/nghttp2# nghttpd -v 8080 ~/key.pem ~/cert.pem
Enter PEM pass phrase:
IPv4: listen 0.0.0.0:8080
IPv6: listen :::8080
```

Côté client, on peut lancer le benchmarking avec la commande suivante:

A terminal window titled 'root@kali: ~/Downloads/nghttp2/src' showing the command 'h2load -n1000 -c150 -m50 https://192.168.43.90:8080' being executed. The terminal window is part of a desktop environment with a taskbar at the top showing various icons and system status (07:17 PM, 95% battery).

```
root@kali: ~/Downloads/nghttp2/src
File Actions Edit View Help
root@kali: ~/Downloads/nghttp2/src# h2load -n1000 -c150 -m50 https://192.168.43.90:8080
```

L'option -n permet de préciser le nombre de requêtes, l'option -c le nombre de clients, et -m le nombre de streams maximums par clients.

Les résultats sont:



```
root@kali: ~/Downloads/nghttp2/src
File Actions Edit View Help
root@kali: ~/...s/nghttp2/src

root@kali:~/Downloads/nghttp2/src# h2load -n1000 -c150 -m50 https://192.168.43.90:8080
starting benchmark ...
spawning thread #0: 150 total client(s). 1000 total requests
TLS Protocol: TLSv1.3
Cipher: TLS_AES_256_GCM_SHA384
Server Temp Key: ECDH P-256 256 bits
Application protocol: h2
progress: 10% done
progress: 20% done
progress: 30% done
progress: 40% done
progress: 50% done
progress: 60% done
progress: 70% done
progress: 80% done
progress: 90% done
progress: 100% done

finished in 1.57s, 637.59 req/s, 118.86KB/s
requests: 1000 total, 1000 started, 1000 done, 0 succeeded, 1000 failed, 0 errored, 0 timeout
status codes: 0 2xx, 0 3xx, 1000 4xx, 0 5xx
traffic: 186.43KB (190900) total, 17.87KB (18300) headers (space savings 85.70%), 147.46KB (151000) data

min      max      mean      sd      12.5% +/- sd
time for request: 265.49ms 1.13s 643.27ms 195.11ms 69.40%
time for connect: 292.43ms 774.43ms 499.25ms 125.55ms 62.67%
time to 1st byte: 560.68ms 1.56s 1.14s 285.78ms 54.00%
req/s      : 3.92 12.48 6.28 1.96 67.33%

root@kali:~/Downloads/nghttp2/src#
```

```
root@kali: ~/Downloads/nghttp2/src
File Actions Edit View Help
root@kali: ~/...s/nghttp2/src x
root@kali:~/Downloads/nghttp2/src# h2load -n2000 -c500 -m100 https://192.168.43.90:8080
starting benchmark ...
spawning thread #0: 500 total client(s). 2000 total requests
TLS Protocol: TLSv1.3
Cipher: TLS_AES_256_GCM_SHA384
Server Temp Key: ECDH P-256 256 bits
Application protocol: h2
progress: 10% done
progress: 20% done
progress: 30% done
progress: 40% done
progress: 50% done
progress: 60% done
progress: 70% done
progress: 80% done
progress: 90% done
progress: 100% done

finished in 8.04s, 248.66 req/s, 48.44KB/s
requests: 2000 total, 2000 started, 2000 done, 0 succeeded, 2000 failed, 0 errored, 0 timeout
status codes: 0 2xx, 0 3xx, 2000 4xx, 0 5xx
traffic: 389.65KB (399000) total, 47.85KB (49000) headers (space savings 80.86%), 294.92KB (302000) data

min      max      mean      sd      12.5% +/- sd
time for request: 973.43ms 4.83s 2.40s 376.23ms 79.20%
time for connect: 1.29s 6.88s 2.74s 792.95ms 79.40%
time to 1st byte: 3.49s 8.03s 5.14s 933.95ms 75.40%
req/s : 0.50 1.15 0.81 0.15 76.00%
```

```
root@kali: ~/Downloads/nghttp2/src
File Actions Edit View Help
root@kali: ~/...s/nghttp2/src x
root@kali:~/Downloads/nghttp2/src# h2load -n10000 -c1000 -m100 https://192.168.43.90:8080
starting benchmark ...
spawning thread #0: 1000 total client(s). 10000 total requests
TLS Protocol: TLSv1.3
Cipher: TLS_AES_256_GCM_SHA384
Server Temp Key: ECDH P-256 256 bits
Application protocol: h2
progress: 10% done
progress: 20% done
progress: 30% done
progress: 40% done
progress: 50% done
progress: 60% done
progress: 70% done
progress: 80% done
progress: 90% done
progress: 100% done

finished in 19.36s, 516.52 req/s, 94.12KB/s
requests: 10000 total, 10000 started, 10000 done, 0 succeeded, 10000 failed, 0 errored, 0 timeout
status codes: 0 2xx, 0 3xx, 10000 4xx, 0 5xx
traffic: 1.78MB (1866000) total, 148.44KB (152000) headers (space savings 88.12%), 1.44MB (1510000) data

min      max      mean      sd      +/- sd
time for request: 16.23ms 13.59s 5.54s 767.38ms 95.10%
time for connect: 2.23s 19.30s 4.85s 1.74s 65.20%
time to 1st byte: 6.57s 19.32s 10.39s 2.03s 61.90%
req/s : 0.52 1.52 1.00 0.19 61.40%
```

### III - Extensions Client Hello

Plusieurs extensions sont déjà présentes sur le client Hello. On a par exemple, le cookie

```
struct {  
    opaque cookie<1..2^16-1>;  
} Cookie;
```

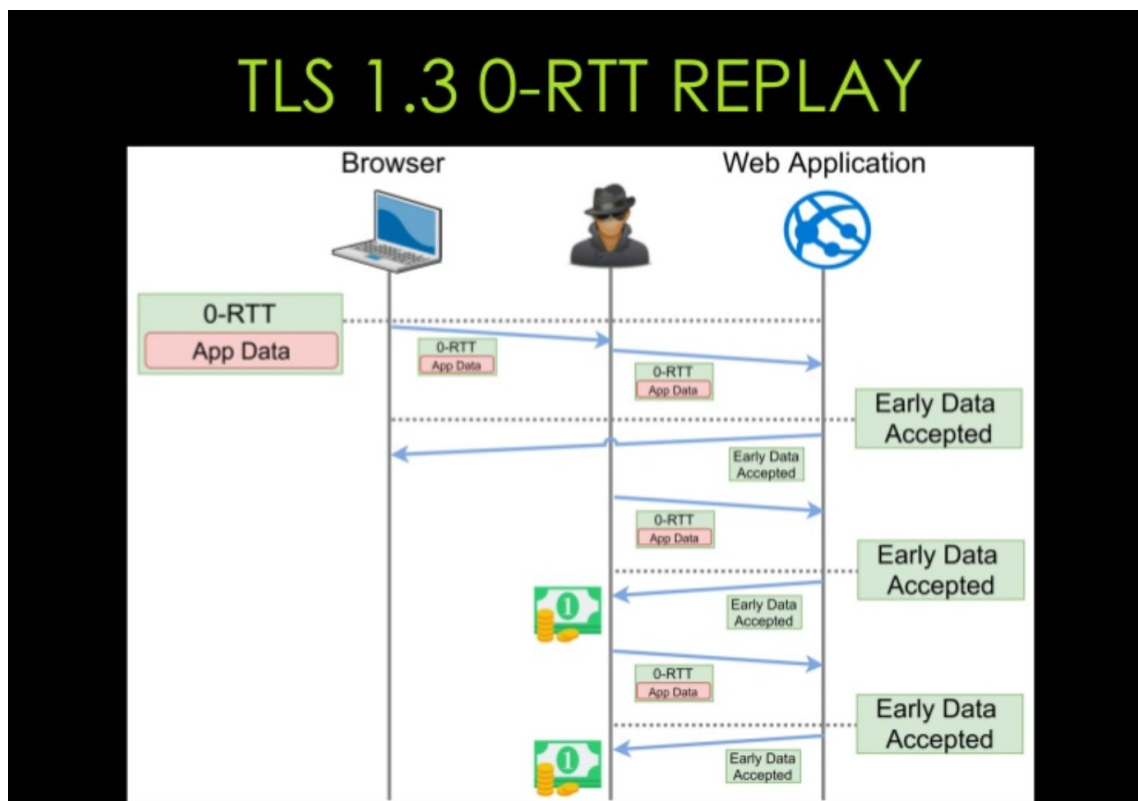
Celui ci permet de prévenir des attaques types Dos en vérifiant la joignabilité de l'adresse IP qu'on prétend avoir.

Remarquons que ce type de protection est aussi présent dans le protocole QUIC par le champ STK: le but est le même: vérifier qu'on a bien l'adresse IP qu'on prétend avoir.

On a aussi des extensions permettant au client de monter quelles versions de TLS sont supportées, quels algorithmes de signatures sont supportés, quelles CA sont acceptées,.... Ainsi, les extensions peuvent permettre soit de rajouter un service facultatif (aider le serveur à sélectionner tel certificat parmi tous ceux qu'il a), et d'autres permettent de rajouter un service de sécurité (le cookie pour limiter les attaques DoS).

Maintenant, il s'agit de proposer d'autres extensions du client Hello. Pour cela, nous nous sommes penchés sur le 0-RTT. Ce mode, très controversé, permet à un client d'envoyer en même temps que le client hello, des données applicatives cryptées par le PSK de la dernière session. Le gain de temps est très important: 1 RTT, c'est l'argument de vente principale de ce mode.

Cependant, cette augmentation de performances vient au coût d'une sécurité affaiblie. En effet, ces 0-RTT sont vulnérables à des attaques de type replay. En voici une:



Un attaquant se met en MITM, le client envoie un 0-RTT que l'attaquant laisse passer. Le serveur répond Early Data Accepted que l'attaquant transfère au client pour lui donner l'illusion qu'il n'y a pas eu de problème. Par la suite, l'attaquant renvoie plusieurs fois le même 0-RTT au serveur qui traite alors la même requête. L'attaquant bloque les Early data accepted suivantes.

Ces requêtes dans le App Data peuvent être des POST \$10,000, le client se mettrait alors à verser plus que prévu.

Le problème est que la dernière version de TLS ne supporte toujours pas de mécanisme anti-replay pour les 0-RTT ! En effet, dans le draft 28 de l'IETF, on peut lire :

“IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. This data is **not forward secret**, as it is encrypted solely under keys derived using the offered PSK.
2. There are **no guarantees of non-replay between connections**. Protection against replay for ordinary TLS 1.3 1-RTT data is provided via the server's Random value, but 0-RTT data does not depend on the ServerHello and therefore has weaker guarantees. This is especially relevant if the data is authenticated either with TLS client authentication or inside the application protocol. The same warnings apply to any use of the early\_exporter\_master\_secret. ”

Le 0-RTT n'est pas forward secret par nature du 0-RTT: on utilise une clé qu'on a déjà. De plus, il n'y a pas de garanties de non-replay du 0-RTT entre connexions.

Nous nous basons dessus pour proposer des extensions au client hello:

### 1) Extension replay protection

Nous proposons une extension au Client Hello **replay protection** qui servira de champ permettant au serveur de savoir si ce 0-RTT est un replay ou pas.

Pour cela, on rajoute déjà une autre extension Client Hello qui est: **anti-replay policy**. Permettant d'indiquer au serveur la politique d'anti replay choisie. Ce champ peut prendre plusieurs valeurs:

- Single Use Ticket :

Autoriser un PSK une seule fois. Le serveur garde en mémoire tous les PSK valides et les détruits dès qu'ils sont utilisés une fois. Le serveur doit aussi implémenter un expiry date pour ces PSK.

- Client Hello Recording

Dériver du Client Hello et du PSK une valeur unique permettant de rejeter les duplicatas. On peut proposer

$$CH_{\text{recording}} = \text{clientHello.random} \parallel H(\text{clientHello.random}, \text{PSK})$$

Le serveur garde alors un strike register pour garder en mémoire si une valeur a déjà été utilisée. Il est difficile pour un attaquant de produire un CHrecording car il n'est pas censé connaître le PSK. Cependant attention, il faut que l'espace des CHrecording soit suffisamment grand pour éviter des collisions type anniversaire !

De plus, il est compliqué pour le serveur de garder en mémoire tous ces CHrecording, cela pourrait d'ailleurs être un vecteur d'attaque: saturer la mémoire du serveur en envoyant plusieurs CHrecording. Le draft 28 propose, sans l'implémenter concrètement, que le serveur ne garde en mémoire les CHrecording que sur une fenêtre d'enregistrement. Le serveur calcule le temps estimé de réception et s'il est en dehors de la fenêtre, il le refuse.

Je ne suis pas complètement d'accord avec cette proposition car même en une courte fenêtre de temps, un attaquant peut envoyer plusieurs CHrecording pour saturer le serveur.

Je préfère la proposition d'implémenter des filtres de Bloom côté serveur: une structure de données dite probabiliste et compacte assurant aucun replay accepté mais potentiellement refuser des non-replay.

- Horodatage

Le client envoie dans le client Hello le moment d'envoi. Le serveur peut refuser le 0-RTT s'il date de plus de 200ms. Cependant, les attaquants peuvent être très rapides. De plus, un serveur pourrait refuser des 0-RTT alors qu'il y avait un simple problème de congestion. L'horodatage peut être  $\text{heure\_envoi} || H(\text{heure\_envoi}, \text{PSK})$ .

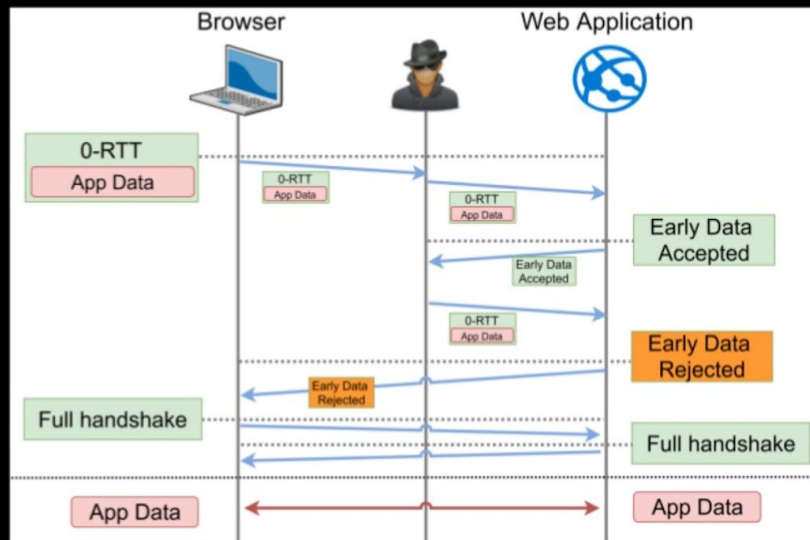
Nous avons proposé 3 valeurs possibles pour l'extension anti-replay policy. Pour chacune de ces politiques, le champ anti-replay protection est rempli de manière adéquat (le ticket dans le cas de Single-use ticket, le CHrecording dans le cas de Client Recording, et un horodatage dans le cas de l'horodatage).

## **2) Extension rejected\_reason**

L'extension replay\_protection est nécessaire, mais non suffisante pour arrêter les attaques basées sur le replay. En effet, voyons l'attaque suivante:



# UNIVERSAL REPLAY ATTACK



Le client envoie un 0-RTT au serveur qui l'accepte. Un attaquant en MITM intercept le early data accepted et rejoue le 0-RTT.

Le serveur refuse celui-ci car il l'a déjà reçu, donc il envoie en Early\_data rejected au client, ce qui le force à refaire un tout nouveau handshake complet.

Puis, le client, croyant que son 0-RTT envoyé a été refusé, décide de renvoyer le contenu Applicatif du 0-RTT.

Le serveur accepte et a donc traité 2 fois le même contenu applicatif !

A premier lieu, on peut se dire que cette attaque montre que la sécurité du 0-RTT doit être assurée au niveau applicatif. Cela ne respecte pas l'indépendance des couches OSI, mais est nécessaire.

Cependant, je propose la chose suivante: le serveur, dans son early\_data\_rejected, pourrait expliquer au client pourquoi il a refusé le 0-RTT. Si le serveur l'a refusé car il a détecté un replay d'une manière ou d'une autre, il faut absolument que le serveur lui dise pour pas que le client ne renvoie le même contenu applicatif par la suite !

Cela pourrait se définir de la façon suivante dans l'attaque:

- Le client envoie un 0-RTT au serveur avec l'extension au CHello:
    - require\_rejected\_reason: True et en précisant le PSK\_binder = PSK\_binder\_c
  - Le serveur répond par un Early\_data\_accepted qui est intercepté par l'attaquant
  - L'attaquant renvoie le 0-RTT au serveur.
  - Le serveur répond
    - Early\_data\_refused
      - > cause: already received
      - Already\_received\_psk\_binder
        - >PSK\_binder\_c
- au client

- L'attaquant bloque et envoie un 0-RTT modifié au serveur en se faisant passer pour le client
- Le serveur répond
  - Early\_data\_refused
    - > cause: failed to decrypt
  - Already\_received\_psk\_binder
    - >PSK\_binder\_c
- A un moment, l'attaquant laisse passer vers le client.
- Le client comprend que son 0-RTT a été accepté une fois, qu'un attaquant a tenté de le rejouer vers le serveur, puis qu'il a tenté d'envoyer un 0-RTT trafiqué en se faisant passer pour le client. Le client fait donc un full handshake et ne renvoie pas le contenu applicatif déjà envoyé dans le 0-RTT. Ou bien, il continue avec la suite du contenu applicatif sans faire de handshake. A la réception du nouveau contenu applicatif, le serveur peut alors oublier qu'il y a eu replay et ne pas avoir à le réindiquer si jamais il y a encore un replay dans un 0-RTT futur.

L'extension au client hello serait donc: **require\_rejected\_reason (true ou false)**. Si la valeur vaut true, le serveur devra, dans sa réponse au 0-RTT indiquer la **cause de rejet** de 0-RTT si cela arrive ainsi que le PSK binder du message rejoué **s'il y a eu replay depuis la réouverture** de la session par le 0-RTT.

Le client aurait ainsi le choix, soit de poursuivre avec la suite du contenu applicatif, soit de faire un nouveau full handshake, mais de pas renvoyer le contenu applicatif déjà reçu.

Cette cause de rejet peut être: already received, session ticket timeout, failed to decrypt,...

Ces causes de rejets pourraient être codées à l'image des codes HTTP (200 OK, 404 page non trouvée, 401 utilisateur non authentifié,...).

### **Conclusion:**

Nous avons pu vérifier via une étude approfondie du handshake et un benchmarking à échelle moyenne que le protocole TLS1.3 constitue une avancée majeure en terme de performance et de services proposés par rapport à TLS1.2. Cependant, l'augmentation des performances se fait souvent au coût d'une sécurité affaiblie. Nous proposons des extensions au client Hello pour y remédier.

### **Bibliographie:**

<https://tools.ietf.org/html/rfc8446>

<https://www.davidwong.fr/tls13/>

<https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/>

<https://nghttp2.org/>

<https://community.centminmod.com/threads/caddy-0-11-5-tls-1-3-http-2-https-benchmarks-part-1.16779/>

<https://fr.slideshare.net/cisoplatfrom7/playback-a-tls-13-story>