

Projet de semestre

SSL/TLS et le support de la *perfect forward secrecy*

Aurélien Coet
Université de Genève
2015

Résumé

Dans le domaine de la sécurité informatique, la notion de *perfect forward secrecy* indique qu'un système ou un protocole permet d'assurer la confidentialité d'échanges chiffrés sur le long terme. SSL/TLS étant l'un des protocoles de sécurité les plus utilisés de nos jours sur internet, ce document s'intéresse au support de la PFS par ce dernier. Après une brève définition de la PFS et du fonctionnement de SSL/TLS, les différentes possibilités offertes par le protocole pour le support de cette propriété sont explorées. Une estimation du taux de déploiement d'implémentations de TLS offrant la PFS est ensuite donnée, sur la base de mesures obtenues dans différents travaux de recherche existant sur le sujet. Enfin, une section de recommandations pratiques décrit les moyens disponibles dans les navigateurs les plus courants pour assurer le support de la PFS côté client, lors de visites sur des sites web utilisant TLS.

1. Introduction :

La sécurité des échanges d'informations sur internet est très difficile à assurer à cause de l'aspect public du réseau. Plusieurs protocoles de sécurité ont donc été mis en place au fil du temps afin de pallier ce problème. Transport Layer Security (TLS) est l'un des plus utilisés de nos jours. Successeur du protocole SSL, développé par Netscape dans les années 90, il a été adopté comme standard depuis RFC 2246 [1].

Situé juste au-dessus de TCP, au niveau de la couche transport, il est implanté dans une majorité de serveurs pour assurer des échanges sécurisés. Il offre des services tels que l'authentification, l'intégrité, la négociation de clés pour des échanges cryptés ainsi que la confidentialité.

En constante évolution, le protocole offre une sécurité relative, mais il reste vulnérable à certains types d'attaques (dont la plupart sont décrites dans [2]). Le protocole étant assez flexible dans sa construction et son implémentation, seule une utilisation de ce dernier avec des paramètres précis permet d'assurer une sécurité qu'il est possible de considérer comme suffisante [3]. Un risque subsiste malgré tout. En particulier, il est possible qu'une communication sûre au moment des échanges effectués entre client et serveur se trouve compromise plus tard, suite à la découverte de clés secrètes par un attaquant. Il est donc nécessaire de remédier à un tel risque, et c'est pour cela qu'a été définie la notion de *perfect forward secrecy* pour des échanges cryptés.

Nous nous intéresserons dans ce travail au support de la *perfect forward secrecy* par TLS. Nous tenterons de définir quelles suites cryptographiques offertes par le protocole permettent d'assurer une telle propriété. Nous verrons quelles précautions en particulier doivent nécessairement être observées afin de s'assurer que des échanges sécurisés avec TLS restent secrets, même après une attaque qui aurait permis de dévoiler les clés privées des intervenants d'un échange.

Ce document se divise en quatre parties : tout d'abord, nous définirons la notion de *perfect forward secrecy* pour des échanges cryptés, et nous décrirons le protocole TLS ainsi que son fonctionnement. Nous verrons ensuite dans la seconde partie du travail quelles sont les suites cryptographiques et les paramètres du protocole qui offrent un support pour la *perfect forward secrecy*. Nous tenterons enfin d'évaluer quel est à ce jour le taux de déploiement d'implémentations de TLS supportant la *perfect forward secrecy* sur internet, puis nous donnerons quelques recommandations sur la façon de paramétrer son navigateur web afin de s'assurer le support de la PFS lors de visites sur des sites utilisant TLS.

2. Définitions :

2.1 La *perfect forward secrecy* [4][5] :

On définit la notion de *perfect forward secrecy* (PFS, parfois traduite « confidentialité persistante » en français) dans le contexte de protocoles d'établissement de clés (ou *key establishment protocols*, noté KEP). Dans de tels protocoles, des algorithmes de cryptographie asymétrique sont très souvent utilisés afin d'authentifier les participants d'échanges sécurisés et de dériver les clés secrètes qui serviront à assurer la confidentialité d'échanges ultérieurs entre eux.

Des clés secrètes dites « principales » sont utilisées sur le long terme pour négocier entre les participants d'autres clés, appelées « clés de session » (aussi secrètes), qui ne servent que sur une durée plus courte à chiffrer les échanges entre participants du protocole. Les clés de session sont renégociées pour chaque nouvelle connexion entre les participants, contrairement aux clés principales qui sont réutilisées à chaque fois.

La *perfect forward secrecy* est la caractéristique qui assure que la découverte par un attaquant des clés principales de l'un ou des deux participants d'échanges cryptés ne compromet pas la confidentialité des clés de session [5].

Cela signifie donc que si un attaquant est capable d'observer et d'enregistrer des échanges cryptés (sans pouvoir les déchiffrer), et qu'il a plus tard accès (soit en cassant l'algorithme de cryptographie asymétrique, soit par d'autres moyens) aux clés privées de l'un ou des deux participants des échanges, il ne pourra pas trouver les clés de session utilisées dans les échanges enregistrés, et sera donc incapable de les décrypter.

La PFS est donc une propriété qui assure la confidentialité sur le long terme des échanges cryptés, et ce malgré une éventuelle faille ultérieure dans le protocole utilisé ou la découverte par une entité extérieure d'un secret qui aurait été utilisé dans les échanges.

Dans [6] et [7], les auteurs soulignent l'importance de la PFS. En effet, TLS est encore vulnérable à de nombreuses attaques, et il existe toujours un risque d'en voir apparaître de nouvelles, comme nous l'a montré en 2014 la faille *Heartbleed*. Pour cette raison, il est impératif de protéger tous les échanges effectués sur internet face à d'éventuelles attaques futures, une propriété que la PFS permet d'assurer.

2.2 Le protocole SSL/TLS [5] [8]:

2.2.1 Historique :

Secure Socket Layer (SSL) est un protocole de sécurité pour les échanges sur internet développé par la compagnie Netscape à partir de 1994. Son successeur, *Transport Layer Security* (TLS), fut défini dans sa première version en 1999 dans RFC 2246 [1] par l'*Internet Engineering Task Force* (IETF), puis amélioré en 2006 dans RFC 4346 [9] (TLS 1.1). Le protocole est fortement inspiré de SSL (il s'agit en réalité d'une amélioration standardisée), et possède donc de nombreuses similarités avec ce dernier, à tel point que les deux sont dans de nombreuses implémentations compatibles l'un avec l'autre. Cette rétro-compatibilité de TLS avec SSL est toutefois fortement déconseillée de nos jours, car SSL possède de nombreuses failles corrigées dans TLS.

Aujourd'hui, TLS a presque totalement supplanté SSL sur internet. Sa version actuelle, TLS 1.2, a été définie en août 2008 dans RFC 5248 [10] (le protocole a encore été amélioré en 2011 dans RFC 6176, avec la suppression du support de la rétro-compatibilité avec SSL).

2.2.2 Architecture du protocole :

TLS se situe juste au-dessus du protocole TCP de la couche transport d'internet (selon le modèle OSI). Il est en fait constitué d'un ensemble de protocoles, qui sont répartis sur deux couches. La première consiste en un unique protocole, le *Record Protocol*, juste au-dessus de TCP. La seconde, encore au-dessus, est une suite de protocoles qui utilisent le *Record Protocol* : on y trouve *http* (appelé *https* lorsqu'il est utilisé avec TLS), le *Handshake Protocol*, le *Change Cipher Spec Protocol*, l'*Alert Protocol* et enfin le *Heartbeat Protocol*. De la seconde couche, seuls le *Handshake Protocol*, *Change Cipher Spec Protocol* et *Alert Protocol* sont considérés comme faisant véritablement partie de TLS. Les autres ne font qu'utiliser certains aspects du *Record Protocol* pour leur propre fonctionnement.

La figure suivante, tirée de [8], illustre bien l'architecture de TLS :

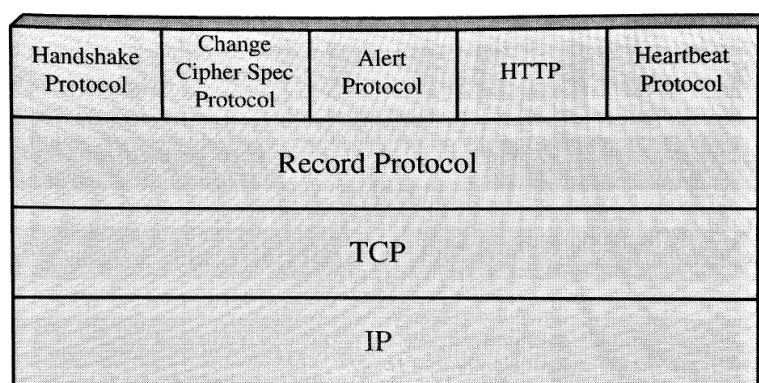


Figure 22.4 SSL/TLS Protocol Stack

2.2.3 Session et connexion :

Deux notions importantes de TLS sont celles de session et de connexion. Ces dernières sont liées aux échanges entre participants du protocole à la fois sur le long et le court terme.

En voici la définition faite dans [8] (traduite depuis l'anglais) :

« *Session* : Une session TLS est une association entre client et serveur. Les sessions sont créées par le Handshake Protocol. Les sessions définissent un ensemble de paramètres de sécurité cryptographique, qui peuvent être partagés sur de multiples connexions. Les sessions sont utilisées afin d'éviter l'opération coûteuse de négociation de nouveaux paramètres de sécurité pour chaque connexion. »

« *Connexion* : Une connexion est un transport (au sens de la définition du modèle OSI des couches) qui offre un type de service approprié. Pour TLS, de telles connexions sont des relations pair-à-pair. Les connexions sont éphémères. Chaque connexion est associée à une session. »

On peut donc voir les sessions comme un ensemble de paramètres cryptographiques négociés entre un client et un serveur sur le long terme, qui peuvent ensuite être réutilisés pour plusieurs connexions. Une session peut donc être associée à de multiples connexions.

Une connexion est en revanche un accord éphémère de paramètres de sécurité entre client et serveur (il s'agit de paramètres tels que des clés symétriques pour l'encryption et le calcul de fonctions de hachage). Une fois les échanges entre les deux parties du protocole terminés, la connexion est interrompue définitivement, alors que la session est sauvegardée. Une connexion utilise les paramètres négociés dans une session pour sa mise en place.

2.2.4 Fonctionnement de TLS :

Handshake Protocol :

La toute première étape, lorsque qu'un client souhaite se connecter à un serveur de façon sécurisée sur internet, consiste à mettre en place une connexion sûre entre les deux parties. Pour ce faire, le *Handshake Protocol* est utilisé, c'est donc le premier protocole de TLS qui a lieu entre client et serveur pour initier les échanges sécurisés qui suivront. Il permet à un client d'authentifier l'identité d'un serveur (et éventuellement l'inverse dans certains cas), puis de négocier les algorithmes cryptographiques pour le chiffrement et le calcul de MAC, ainsi que les clés associées qui seront utilisées dans des échanges à travers le *Record Protocol*.

Le *Handshake Protocol* utilise la notion de cryptographie asymétrique avec clés publiques ainsi que les certificats de type x.509 pour l'authentification des participants des échanges. Comme il est très rare que les clients, dans de tels échanges, possèdent des certificats auprès des autorités de certification, seul le serveur est généralement authentifié. Ceci peut parfois représenter une faille dans le protocole et mener à certains types d'attaques (*Man in the Middle*).

Le *Handshake Protocol* se divise en quatre phases lors desquelles des messages entre client et serveur sont échangés afin de convenir de divers paramètres de sécurité. Ces quatre phases sont les suivantes :

1) Lors de la première phase, le client initie la connexion avec le serveur en lui envoyant un message *client_hello*. Il indique dans ce message des éléments tels que la dernière version de TLS qu'il supporte, un nombre aléatoire généré à l'aide d'un générateur pseudo-aléatoire sûr, ainsi qu'une *session ID* qui indique soit que la connexion doit se faire sur une session déjà existante entre les deux parties, soit qu'une nouvelle session doit être créée. Il indique aussi dans ce message la liste

des algorithmes cryptographiques qu'il supporte et qui sont préférés pour l'échange (ces combinaisons d'algorithmes sont appelées *cipher suites*), ainsi qu'une méthode de compression pour les données à échanger. Le message est envoyé sans être chiffré.

Après la réception du message *client_hello*, le serveur répond avec un message *server_hello* du même type dans lequel il indique au client quels choix il a fait par rapport aux propositions qui lui ont été soumises.

2) Dans la deuxième phase, le serveur envoie au client ses clés publiques ainsi que le certificat correspondant pour s'authentifier si cela lui est demandé. Le serveur envoie ensuite au client dans le message *server_key_exchange* les informations nécessaires pour la génération des clés de session, selon les algorithmes choisis. Il termine la seconde phase avec un message *server_done* qui indique que l'envoi de *server_hello* et des certificats est terminé côté serveur.

3) La troisième phase consiste en la vérification par le client des certificats du serveur ainsi que l'envoi de ses certificats par le client au serveur si cela est nécessaire. Le client envoie ensuite à son tour ses informations pour la génération de clés privées de session dans un message *client_key_exchange*. Suite à l'envoi des messages *server_key_exchange* (facultatif, selon les algorithmes choisis) et *client_key_exchange*, le *master secret* est calculé. Ce dernier est un secret partagé entre client et serveur qui permettra de dériver les différentes clés de session pour la connexion TLS initiée.

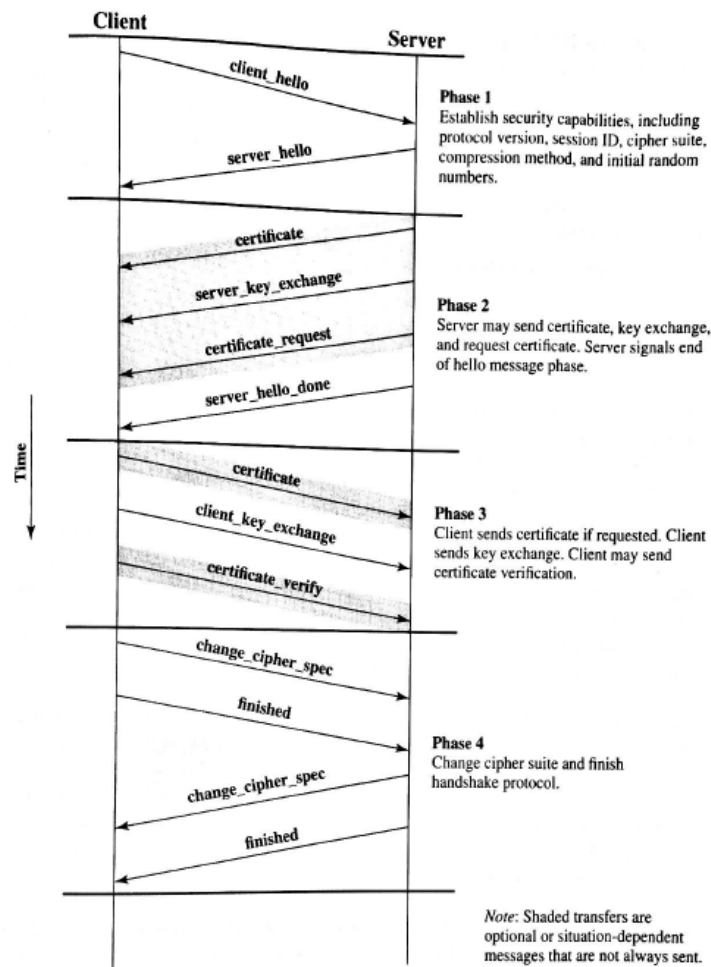
4) Enfin, dans la quatrième phase, le client envoie un message *change_cipher_spec* au serveur afin de lui indiquer que tous les paramètres négociés peuvent être activés pour la connexion. Ce message fait partie du protocole *Change Cipher Spec* (ce protocole ne consiste en fait qu'en un seul message).

Finalement, un message *finished* est envoyé au serveur par le client à travers le *Record Protocol*, avec les nouveaux paramètres qui ont été mis en place.

Le serveur effectue les mêmes opérations dans le sens inverse, et le *Handshake Protocol* est alors terminé.

Nous verrons que le choix des *cipher suites* qui est fait lors du *Handshake Protocol* est d'une importance capitale pour le support de la *perfect forward secrecy* par TLS. En effet, le type d'algorithmes cryptographiques utilisés, en particulier pour la négociation des clés de session, permet de définir si cette notion est respectée pour une connexion donnée.

Le schéma suivant illustre les échanges de messages qui ont lieu lors du *Handshake Protocol* [8] :



Record Protocol :

Le *Record Protocol* est la partie de TLS qui offre des services de confidentialité et d'intégrité. Il sert à l'encapsulation des protocoles qui se trouvent au-dessus de lui.

Lors du *Handshake Protocol*, des paramètres de sécurité sont négociés entre client et serveur afin de définir les clés symétriques pour le chiffrement des messages, ainsi que le calcul de *message authentication codes* (MAC). Le *Record Protocol* utilise ces clés pour opérer de la façon suivante : les données reçues depuis la couche application sont d'abord fragmentées en blocs d'au plus 214 octets, puis éventuellement compressées (selon les paramètres choisis pour la connexion TLS). Le *Record Protocol* calcule ensuite le MAC de ces données à l'aide de la clé et de la fonction de hachage correspondantes et l'ajoute à la fin de chaque fragment. Ceci permet de s'assurer, à la réception des fragments, de leur intégrité.

Une fois le MAC ajouté, le protocole chiffre les fragments avec sa clé symétrique d'encryption, puis il ajoute au début de chacun d'entre eux un en-tête indiquant la version du protocole et des informations sur la taille des blocs. C'est ainsi qu'est obtenue la confidentialité des échanges.

Chaque bloc obtenu suite à ces opérations est envoyé au destinataire qui, à leur réception,

effectue les opérations inverses, puis ré-assemble l'ensemble des segments afin d'obtenir les données de départ qu'il peut transmettre à la couche application.

Alert Protocol et Heartbeat Protocol :

L'*Alert Protocol* permet au client ou au serveur d'envoyer à son correspondant des messages d'alerte lors d'une connexion TLS. Ces alertes peuvent par exemple servir à indiquer que la connexion va être terminée, soit parce que la sécurité de cette dernière est compromise, soit pour une autre raison. Un autre exemple d'alerte permet de signaler un MAC incorrect pour un message.

Le *Heartbeat Protocol* permet à un intervenant dans le protocole TLS d'envoyer un message *heartbeat_request* à son correspondant afin de s'assurer qu'il est toujours actif sur la connexion. Le correspondant répond alors avec un message *heartbeat_response* si c'est le cas. Ceci permet de maintenir la connexion même lorsqu'elle est inactive durant une certaine période de temps, ainsi que de générer du trafic sur la connexion pour éviter qu'elle ne soit bloquée par d'autres systèmes de sécurité extérieurs (tels que des *firewalls* par exemple).

Ces deux protocoles ne nous intéresseront pas pour la *perfect forward secrecy*, car ils ne jouent pas de rôle réel vis-à-vis de cette dernière.

2.3 Les *cipher suites* offertes par TLS [10]:

Comme indiqué plus haut, TLS offre plusieurs suites cryptographiques (ou *cipher suites*) parmi lesquelles client et serveur peuvent choisir lors d'une connexion. Ces différentes suites consistent en des combinaisons d'algorithmes de cryptographie symétrique et asymétrique ainsi que de fonctions de hachages, afin d'offrir des services de sécurité tels que l'authentification (cryptographie asymétrique et certificats), l'intégrité (fonctions de hachage et MACs), ainsi que la confidentialité (algorithmes de cryptographie symétrique). La négociation de clés de session se fait à l'aide de la cryptographie asymétrique.

Pour l'authentification, TLS utilise RSA ou DSS. Pour la confidentialité, divers algorithmes tels que 3DES, AES (128 ou 256 bits) ou RC4 sont disponibles (bien que l'utilisation de RC4 soit à ce jour déconseillée à cause des failles qui touchent l'algorithme). Il est possible de choisir parmi des fonctions de hachage comme MD5, SHA-1 ou SHA 256 pour l'intégrité.

Pour la négociation des clés de session, TLS offre plusieurs possibilités :

1) La première est l'utilisation de RSA seul : dans ce cas la même clé publique est utilisée pour l'authentification du serveur (éventuellement du client) et pour le calcul du *master secret*. Le client authentifie le serveur à l'aide du certificat et de la clé publique de ce dernier. Il envoie ensuite au serveur une suite de 48 octets appelée *pre-master secret*, chiffrée avec la clé publique authentifiée précédemment. Le master secret est alors calculé à partir de cette valeur et des nombres aléatoires échangés dans les messages *client* et *server hello*.

2) Une autre possibilité est l'utilisation de l'algorithme de Diffie-Hellman (DH), couplé avec DSS ou RSA. Dans ce cas, DH est utilisé pour la négociation des clés de session, et RSA ou DSS (au choix) sert à l'authentification. Nous détaillerons plus loin le fonctionnement de DH. Il existe aussi une version de Diffie-Hellman sur les courbes elliptiques : *Elliptic Curve Diffie-Hellman* (ECDH).

3) Il existe enfin pour la négociation des clés deux autres algorithmes pouvant être utilisés comme DH : *Diffie-Hellman Ephemeral* (DHE) et *Elliptic Curve Diffie-Hellman Ephemeral* (ECDHE). Nous verrons plus loin le fonctionnement de ces derniers et leurs différences avec DH.

Pour une liste détaillée des combinaisons exactes d'algorithmes (*cipher suites*) disponibles dans TLS, se référer à [10], appendice A.5. Nous nous intéresserons principalement dans ce document aux algorithmes utilisés pour la négociation des clés de session, car ce sont eux qui permettent de définir si la PFS est respectée lors d'échanges via TLS.

3. SSL/TLS et la PFS :

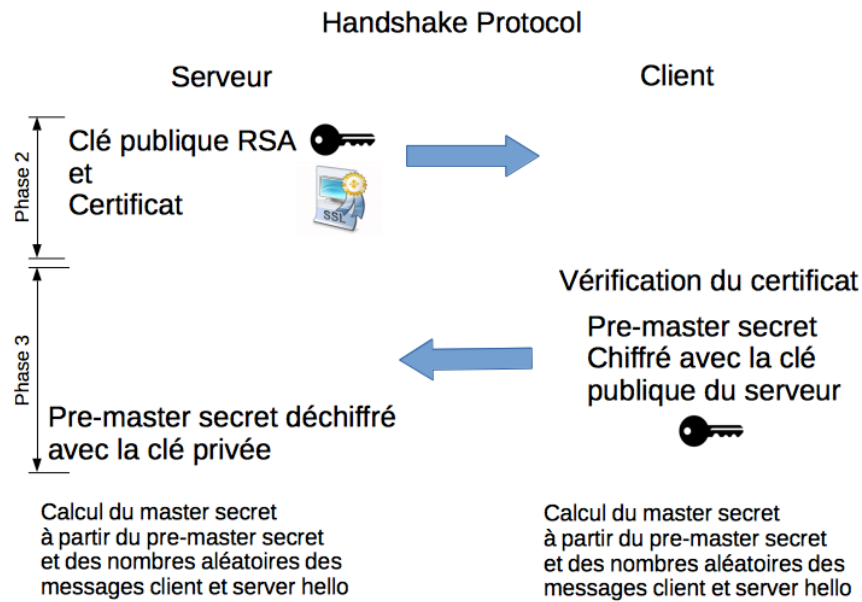
Comme nous l'avons mentionné plus haut, plusieurs *cipher suites* sont disponibles dans TLS pour assurer les divers services de sécurité offerts par le protocole. En particulier, nous avons vu qu'il existe cinq algorithmes différents pour l'échange de clés : RSA, Diffie-Hellman, ECDH, DHE et ECDHE. Nous allons dans la suite étudier chacun de ces algorithmes afin de montrer lesquels offrent un support pour la *perfect forward secrecy* et quels sont les avantages et inconvénients de chacun.

3.1 RSA [11] :

Lorsque RSA est choisi pour l'authentification et l'échange de clés, les opérations suivantes sont effectuées dans le *Handshake Protocol* de TLS :

- 1) Le serveur envoie au client, dans la phase 2 du *Handshake Protocol*, sa clé publique RSA ainsi que le certificat associé.
- 2) Dans la phase 3, le client vérifie l'identité du serveur à l'aide de son certificat, puis il génère une suite de 48 octets appelée *pre-master secret*, qu'il envoie au serveur sous forme chiffrée avec la clé publique de ce dernier dans un message *client_key_exchange*.
- 3) Le serveur déchiffre le message *client_key_exchange* reçu à l'aide de sa clé privée pour retrouver le *pre-master secret*.
- 4) Le *master secret* est calculé par le client et le serveur à partir du *pre-master secret* et des nombres aléatoires des messages *client* et *server hello* de la phase 1 du protocole.

La figure suivante illustre ce protocole d'échange de clés :



On remarque qu'à chaque fois qu'une nouvelle connexion est initialisée entre client et serveur, de nouvelles clés de session sont générées en calculant un nouveau master secret (puisque le *Handshake Protocol* a lieu à chaque nouvelle connexion). Le *pre-master secret* étant chiffré avec la clé privée du serveur, seul le client et lui y ont accès, à condition que personne d'autre ne connaisse cette clé. Si un adversaire enregistre les échanges entre client et serveur mais ne connaît pas la clé privée du serveur, il est incapable de les déchiffrer car il n'a pas accès au *master secret*.

En revanche, si un attaquant venait à accéder à la clé privée du serveur, il pourrait déchiffrer le *pre-master secret* envoyé dans le message *client_key_exchange* et calculer le *master secret* car les nombres aléatoires des messages *client* et *serveur hello* sont échangés en clair entre les participants. Cela implique que, comme le serveur utilise toujours la même paire de clés publique et privée dans les échanges, la PFS n'est pas respectée. En effet, imaginons qu'un adversaire enregistre tous les échanges entre client et serveur d'une session TLS utilisant RSA pour l'échange de clés, et ce dans chacune de ses connexions. Imaginons aussi que l'adversaire vienne à connaître la clé privée du serveur, soit en la volant, soit en cassant l'algorithme RSA (ce qui reste peu probable dans les années à venir pour des clés de taille suffisante, au vu de la difficulté du problème mathématique de factorisation associé à l'algorithme). Alors, il serait capable de déchiffrer tous les messages *client_key_exchange* des échanges enregistrés, et donc de retrouver tous les *master secrets* correspondants. Ceci lui donnerait la capacité de déchiffrer tous les messages précédemment échangés avec le *Record Protocol* entre client et serveur pour la session observée. La PFS n'est donc pas assurée ici.

3.2 Diffie-Hellman :

3.2.1 L'algorithme de Diffie-Hellman [5] [11] :

Diffie-Hellman est un algorithme d'échange de clés basé sur la cryptographie publique, introduit en 1976 par Whitfield Diffie et Martin Hellman et inspiré d'une idée du cryptologue Ralph

Merkle. L'algorithme se base sur les groupes multiplicatifs et les générateurs, et il permet la construction sur un canal non confidentiel d'une clé secrète partagée entre deux entités.

Nous allons voir ici le fonctionnement général de l'algorithme. Deux entités A et B souhaitent construire une clé secrète partagée. Pour ce faire, les étapes suivantes ont lieu :

- 1) Un grand nombre premier p est généré, ainsi que α un générateur de Z_p^* (c'est-à-dire que pour tout $x \in Z_p^*$, il existe k tel que $\alpha^k \equiv x \pmod p$, avec Z_p^* le groupe multiplicatif de Z_p). Les deux nombres sont rendus publics pour les deux parties de l'échange.
- 2) A génère un nombre aléatoire x qu'il garde secret et calcule $\alpha^x \pmod p$. Il envoie le résultat à B.
- 3) B génère un nombre aléatoire y qu'il garde lui aussi secret, et il calcule $\alpha^y \pmod p$. Il envoie le résultat à A.
- 4) A calcule $(\alpha^y)^x \pmod p = K$ la clé secrète, et B calcule de son côté $(\alpha^x)^y \pmod p = K$. A et B sont donc capables de calculer la même clé K à partir de l'information transmise par l'autre et de leur propre secret.

On appelle paramètres publics de Diffie-Hellman les valeurs $\alpha^x \pmod p$ et $\alpha^y \pmod p$ correspondant aux secrets x et y des participants de l'algorithme, ainsi que p et α . Afin de connaître K, il faudrait qu'un attaquant soit capable de trouver $\alpha^y \pmod p$ à partir de $\alpha^x \pmod p$ ou $\alpha^y \pmod p$. Ceci est appelé le *Diffie-Hellman Problem* (DHP) et a été prouvé impossible à résoudre en un temps raisonnable pour des nombres suffisamment grands. En effet, ce problème est aussi difficile à résoudre qu'un problème mathématique réputé difficile appelé *problème des logarithmes discrets sur un groupe fini*. Diffie-Hellman appartient donc à la catégorie *provable security* d'algorithmes. L'algorithme est sûr pour l'établissement de clés à condition qu'un attaquant ne connaisse pas le secret x ou y d'un des participants de l'algorithme.

Comme indiqué précédemment, l'algorithme de Diffie-Hellman est utilisé sous différentes formes dans TLS, que nous allons détailler dans la suite.

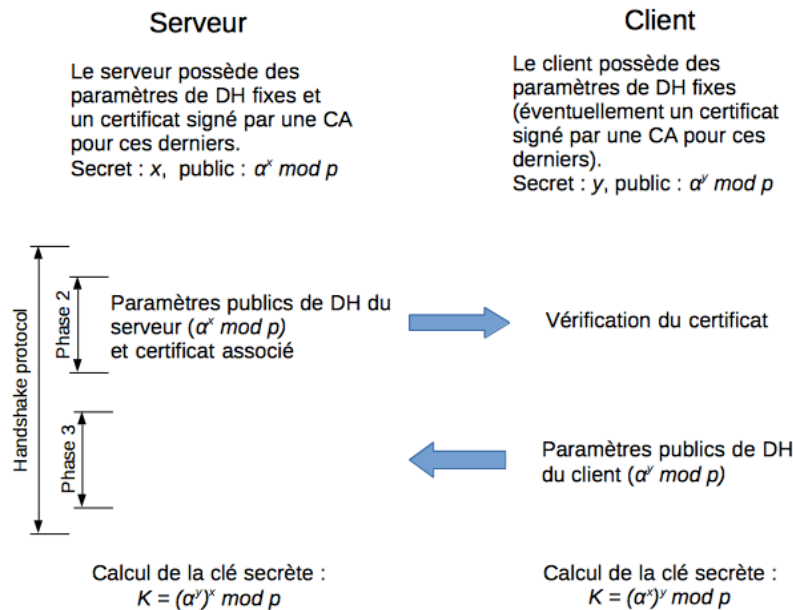
3.2.2 Fixed Diffie-Hellman (DH) :

On parle de *Diffie-Hellman* ou *Fixed Diffie-Hellman* (par opposition à *DH Ephemeral* ou DHE) lorsque les mêmes paramètres de Diffie-Hellman sont utilisés pour toutes les connexions. Dans ce cas, le client et le serveur possèdent tous deux des secrets x et y qu'ils sauvegardent et réutilisent pour tous leurs échanges dans DH. Les mêmes paramètres publics de Diffie-Hellman sont donc utilisés pour la négociation des clés de session entre client et serveur (on entend par paramètres publics les valeurs $\alpha^x \pmod p$ et $\alpha^y \pmod p$ associés aux secrets x et y que les participants se communiquent en clair).

Dans TLS, *Fixed DH* est implémenté comme suit [12] : lors de la phase 2 du *Handshake Protocol*, le serveur communique au client un certificat contenant ses paramètres publics de DH signés par une autorité de certification. Le client communique à son tour dans la phase 3 ses paramètres publics au serveur, éventuellement signés eux aussi par une CA si l'authentification côté client est demandée. Client et serveur peuvent alors calculer chacun de leur côté la clé secrète pour les échanges.

Fixed Diffie-Hellman

Valeurs p et α un générateur de \mathbb{Z}_p^* communes entre client et serveur.



Le fonctionnement de Fixed DH dans TLS.

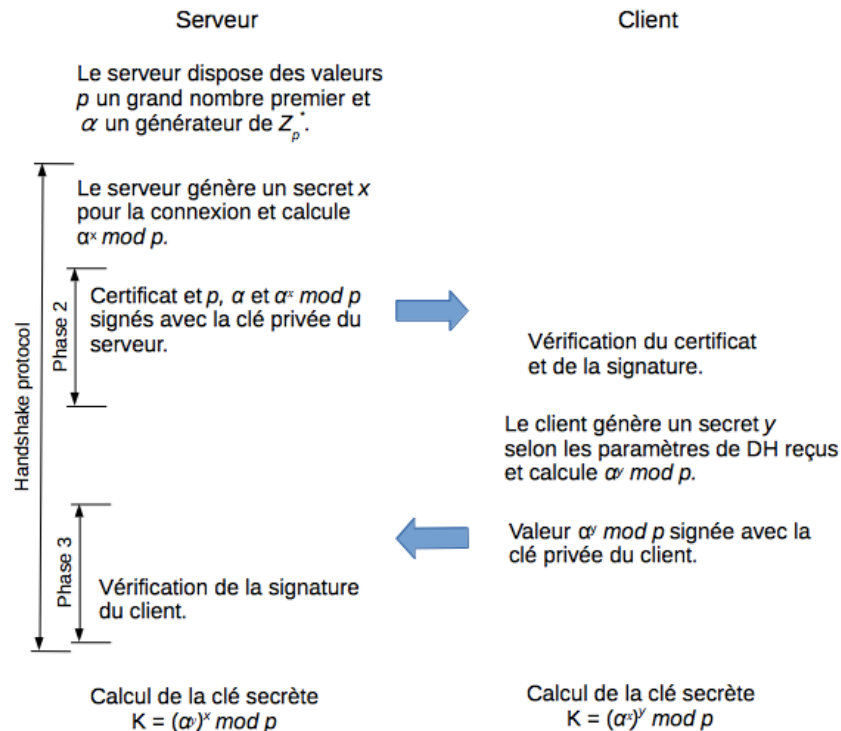
On remarque ici que le même secret est toujours utilisé entre les deux participants (tant qu'ils ne changent pas leurs paramètres de Diffie-Hellman auprès de la CA), puisque le calcul du secret partagé est une opération déterministe pour des mêmes paramètres. La *perfect forward secrecy* n'est donc pas supportée pour des connexions utilisant une *cipher suite* avec DH pour l'échange de clés. En effet, si un adversaire trouve le paramètre secret de DH soit du client, soit du serveur qui participent à une session TLS, il est alors capable de calculer le secret partagé entre les deux sur le long terme et donc de déchiffrer tous leurs échanges préalables. Pour cette raison, une autre utilisation de Diffie-Hellman appelée *Ephemeral Diffie-Hellman* (DHE) a été introduite dans TLS.

3.2.3 Ephemeral Diffie-Hellman :

Dans DHE, l'algorithme de Diffie-Hellman est utilisé avec des paramètres dits éphémères. De nouveaux secrets x et y pour les participants A et B de l'algorithme sont donc générés pour chaque nouvel échange entre les deux parties et jamais réutilisés.

L'implémentation dans TLS de cet algorithme est la suivante [12] : à chaque nouvelle connexion TLS, le client et le serveur génèrent de nouveaux paramètres de Diffie-Hellman. Lors du *Handshake Protocol*, le serveur, qui est en possession d'un grand nombre premier p ainsi que d'un générateur de \mathbb{Z}_p^* α , génère un nouveau secret x et calcule $\alpha^x \bmod p$. Il communique ensuite au client son certificat et sa clé publique (selon l'algorithme choisi), puis il lui envoie, signés avec la clé privée correspondante, les paramètres de Diffie-Hellman suivants : p , α et $\alpha^x \bmod p$. Le client génère de son côté un secret y et calcule $\alpha^y \bmod p$, puis il envoie au serveur sa clé publique et ses paramètres de Diffie-Hellman signés avec sa clé privée (avec éventuellement un certificat pour sa clé publique si cela est demandé). Les deux peuvent alors calculer le secret partagé qui servira pour la connexion.

Ephemeral Diffie-Hellman



Le fonctionnement de DHE dans TLS.

Comme des paramètres différents sont utilisés pour chaque connexion, le secret partagé entre client et serveur sera à chaque fois différent. Si un adversaire était capable de trouver le paramètre secret de DH de l'un des participants, il pourrait donc dériver la clé secrète pour la connexion correspondante uniquement, et non pour les précédentes ou les suivantes. La *perfect forward secrecy* est donc ici respectée. Un autre avantage de DHE est que des clés différentes sont utilisées pour l'authentification et la négociation des clés de session.

Bien que DHE présente l'avantage d'offrir la *perfect forward secrecy* dans TLS, ce dernier présente un souci majeur de temps de calcul lors de l'échange des clés dans le *Handshake Protocol*. En effet, afin d'avoir une sécurité jugée suffisante, il est nécessaire que les paramètres utilisés dans Diffie-Hellman soient d'une taille suffisamment grande ([11] en général, la même taille que celle de la clé privée utilisée pour les signer, soit 2048 bits à ce jour pour RSA par exemple). Cela implique un temps de calcul important à la fois du côté du client et du serveur, puisque tous deux doivent effectuer des calculs exponentiels sur de très grands nombres pour dériver la clé secrète. La négociation de clés de session avec Diffie-Hellman est donc beaucoup plus lente qu'avec RSA, comme le montrent dans [13] les mesures effectuées par les auteurs. Heureusement, il existe une solution à ce problème : l'utilisation de Diffie-Hellman sur des courbes elliptiques, beaucoup plus rapide que l'utilisation de DH ou que RSA.

3.3 Elliptic Curve Diffie-Hellman (ECDH) :

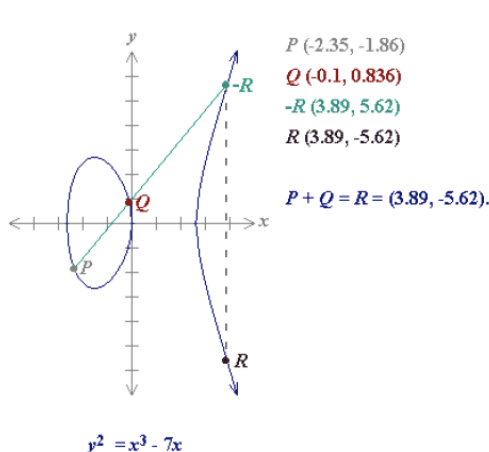
3.3.1 Les courbes elliptiques [14] [5]:

Les courbes elliptiques sont un objet mathématique qui fut introduit dans le monde de la cryptographie en 1985 par Neal Koblitz et Victor Miller (de façon indépendante), mais le sujet était déjà connu et étudié depuis plusieurs siècles par les mathématiciens. La structure riche et complexe de telles courbes permet de nombreuses applications en cryptographie et en sécurité informatique. Le développement de la cryptographie à courbes elliptiques observé à l'heure actuelle est principalement dû au fait que des clés de petite taille peuvent être utilisées sans faire de concessions sur la qualité de la sécurité des systèmes, ce qui implique des temps de calculs faibles pour les algorithmes utilisant de tels objets.

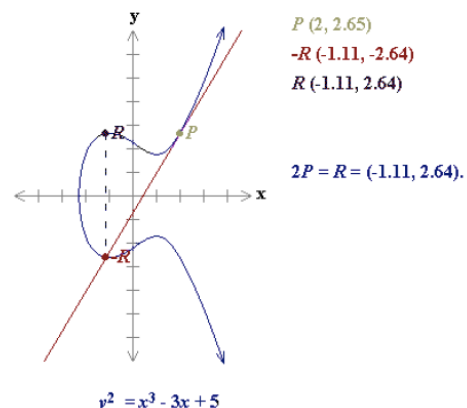
Sans trop entrer dans les détails mathématiques, car le sujet est très vaste et complexe, nous allons décrire ici le fonctionnement des courbes elliptiques.

Une courbe elliptique est un ensemble de points E satisfaisant l'équation de la forme $Y^2 = X^3 + aX + b$, avec la condition supplémentaire que $X^3 + aX + b$ ne possède pas de racines multiples, c'est-à-dire que le discriminant de l'équation de la courbe $4a^3 + 27b^2 \neq 0$. Une courbe elliptique possède un point à l'infini, noté P_∞ , qui peut être vu « *comme un unique point infiniment loin le long de l'axe y où la courbe se 'compactifie' [au sens mathématique du terme]* » (traduit de l'anglais depuis [14]). Le point P_∞ est l'élément identité de la courbe E [5] : pour un point $P := (x, y) \in E$, on définit $-P := (x, -y) \in E$ et $P + (-P) = P_\infty$.

Des opérations d'addition et de multiplication par un scalaire sont disponibles sur les courbes elliptiques. L'addition est définie comme suit [5] : pour deux points P et Q tels que $Q \neq -P$, on définit $P + Q = R \in E$, où $-R$ est le troisième point d'intersection entre la courbe E et la droite passant par P et Q . La multiplication d'un point $P \in E$ par 2, c'est-à-dire la somme de P avec lui-même, donne $2P = R$, où $-R$ est le point d'intersection entre E et la droite tangente à la courbe E au point P [5]. La multiplication d'un point $P \in E$ par un scalaire k est la somme de P avec lui-même k fois.



[5] : addition de deux points sur une courbe elliptique.



[5] : multiplication par 2 d'un point P sur une courbe elliptique.

Pour une courbe elliptique E définie sur un corps Z_p , avec p un grand nombre premier, le calcul du scalaire $k \in Z_p$ tel que $kP = Q$, pour P, Q donnés, est reconnu comme un problème mathématique très difficile appelé *Elliptic Curve Discrete Logarithm Problem* (ECDLP). C'est sur

la difficulté de ce problème que se base la sécurité de l'algorithme de Diffie-Hellman sur des courbes elliptiques.

3.3.2 Elliptic Curve Diffie-Hellman (ECDH) :

Les courbes elliptiques ont été introduites dans TLS depuis RFC 4492 ([15]) comme une extension du protocole. Elles sont utilisées pour l'échange de clés entre client et serveur ainsi que pour l'authentification. En particulier, l'algorithme de Diffie-Hellman sur les courbes elliptiques (ECDH) a été ajouté aux options disponibles pour l'échange de clés dans le standard. Pour l'authentification, *Elliptic Curve Digital Signature Algorithm* (ECDSA) a été inclus dans TLS en plus des autres algorithmes disponibles.

Nous allons nous intéresser ici à l'algorithme de Diffie-Hellman sur les courbes elliptiques. Ce dernier existe soit sous forme fixe (ECDH), soit éphémère (ECDHE), tout comme cela est le cas pour Diffie-Hellman.

Dans le cas où ECDH est utilisé, le fonctionnement de l'algorithme dans TLS est le suivant [15] : le serveur participant à la connexion TLS possède une paire de clés publique et privée pour une courbe elliptique E donnée ainsi qu'un point de départ P . Sa clé privée consiste en une valeur entière x , gardée secrète, et sa clé publique en la valeur xP , la multiplication sur la courbe E du point de départ P par l'entier x . Le serveur possède un certificat signé par une autorité de certification contenant la valeur de la clé publique ainsi que des informations sur la courbe E utilisée et le point de départ P . Les étapes qui ont lieu sont alors les suivantes :

- 1) Le serveur envoie au client de l'échange son certificat contenant sa clé publique xP ainsi que les informations sur la courbe et le point de départ utilisé.
- 2) Une fois le certificat reçu, le client vérifie sa validité et génère une paire de clés y et yP sur la même courbe que le serveur. Il envoie ensuite au serveur sa clé publique yP et calcule de son côté la valeur $K = yxP$, à partir de son secret y et de la clé publique xP du serveur. La valeur K fait office de *pre-master secret* dans l'échange.
- 3) Une fois la clé publique du client reçue, le serveur peut à son tour calculer $K = xyP$. Client et serveur sont alors tous deux en possession du même *pre-master secret*, à partir duquel ils pourront dériver le *master secret*.

Comme nous l'avons expliqué précédemment dans la définition des courbes elliptiques, il est impossible de retrouver les valeurs x ou y à partir des nombres xP et yP échangés en clair entre les participants. Pour cette raison, un attaquant est incapable de retrouver $K = xyP$ à partir de xP et yP , puisqu'il ne connaît pas x et y . L'échange est donc sûr tant que personne d'autre que le client et le serveur ne connaissent les clés privées x et y .

Toutefois, pour les mêmes raisons que dans *Fixed Diffie-Hellman*, ECDH ne supporte pas la PFS. En effet, la même paire de clés publique et privée est utilisée pour toutes les connexions par le serveur. Ainsi, si un attaquant observe et enregistre tous les échanges d'une session TLS et qu'il parvient à récupérer la clé privée du serveur, il sera capable, pour toutes les connexions de la session, de calculer le *pre-master secret* $K = xyP$, puisqu'il sera en possession de la valeur secrète x utilisée dans chaque échange. L'attaquant pourra alors facilement trouver le *master secret*, la seule partie secrète nécessaire pour son calcul étant le *pre-master secret*. À partir de là, il pourra

déchiffrer tous les échanges qu'il aura enregistré. Il est donc impératif, pour que la PFS soit supportée par le protocole, de ne pas utiliser ECDH pour l'échange de clés.

3.3.3 Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) :

Tout comme dans le cas de Diffie-Hellman, une version dite éphémère de l'algorithme ECDH a été mise en place afin d'assurer la *perfect forward secrecy* : celle-ci s'appelle *Elliptic Curve Diffie-Hellman Ephemeral* (ECDHE). Son fonctionnement est le suivant [11] :

- 1) Tout d'abord, une courbe elliptique E d'équation $Y^2 = X^3 + aX + b$ et respectant les conditions mentionnées plus haut est sélectionnée, ainsi qu'un nombre premier p et un point de départ P . De telles valeurs sont appelées les paramètres publics de ECDH, et leur choix est négocié durant l'échange des messages *client* et *server hello*. Il existe une liste de courbes et de paramètres standards prédéfinis parmi lesquels peuvent choisir client et serveur afin d'éviter d'avoir à générer eux-mêmes ces éléments (par exemple, on trouve comme standard les courbes NIST P-256, P-384, P-521, etc...).
- 2) Une fois les paramètres publics négociés, le serveur envoie au client sa clé publique pour l'authentification et le certificat associé, selon l'algorithme choisi (RSA, DSS, ECDSA, ...). Il génère ensuite un entier x qu'il garde secret et calcule la valeur xP (la multiplication sur E du point P par x), qu'il envoie au client signée avec sa clé privée dans le message *server_key_exchange*.
- 3) Le client vérifie la validité du certificat du serveur et la signature sur le message *server_key_exchange*. Il génère ensuite à son tour une valeur secrète y et calcule la valeur publique yP , qu'il envoie au serveur (dans le *message client_key_exchange*). Le client calcule finalement la valeur $K = yxP$, qui servira de *pre-master secret* pour la connexion.
- 4) À la réception du message *client_key_exchange*, le serveur calcule à son tour la valeur $K = xyP$. Client et serveur sont alors en possession du même *pre-master secret* K .

Tout comme dans le cas de DHE, des valeurs différentes sont ici utilisées d'une connexion à l'autre par le client et le serveur pour le calcul des clés de session (du *pre-master secret* en réalité). Ainsi, même si un attaquant enregistre toutes les connexions qui ont lieu sur une session TLS et qu'il a plus tard accès au secret utilisé dans l'une d'elles, il ne sera alors capable de déchiffrer que les échanges de cette connexion en particulier et non des autres. La PFS est donc bien supportée par l'algorithme ECDHE, comme cela est le cas pour DHE et pour les mêmes raisons.

Le principal avantage de ECDH ou ECDHE sur DH et DHE est la rapidité d'exécution de l'algorithme et la petite taille des clés utilisées (c'est d'ailleurs la raison principale pour laquelle les courbes elliptiques furent introduites dans TLS). En 2014, dans [13], diverses mesures ont été effectuées sur un grand nombre de serveurs utilisant TLS. L'une de ces mesures a permis de déterminer la taille des paramètres utilisés pour DHE par les serveurs faisant appel à cet algorithme, comme le montre la figure suivante, tirée de [13] :

Table 1. Diffie-Hellman parameter sizes used for ephemeral DH (DHE) key exchange.

Size (bits)	Hosts	Percentage
256	2	0.0
512	96,559	34.0
768	933	0.3
1,024	281,714	99.3
1,544	1	0.0
2,048	859	0.3
3,248	2	0.0
4,096	14	0.0

On observe ici qu'un grand nombre de serveurs utilisent des paramètres d'une taille inférieure à celle de leur clé RSA. Or, comme mentionné précédemment, il est indispensable d'utiliser des paramètres d'une taille au moins égale à celle actuellement recommandée pour des clés RSA, soit 2048 bits, afin d'assurer la sécurité de l'échange de clés avec DHE. Une très grande majorité des serveurs utilisant DHE fait donc appel à un système faiblement sécurisé pour son échange de clés : plus de 99 % d'entre eux se servent de paramètres d'une longueur de 1024 bits.

En comparaison, la majorité des serveurs faisant appel à ECDHE pour l'échange de clés utilise la courbe elliptique *secp256r1*, comme le montre la figure suivante ([13]) :

Table 2. Elliptic curves used for Elliptic Curve Diffie-Hellman key exchange.

Curve	Hosts	Percentage
<i>secp256r1</i>	81,789	96.1
<i>sect233r1</i>	3,123	3.6
<i>sect571r1</i>	316	0.3
<i>secp384r1</i>	86	0.1
<i>secp521r1</i>	73	0.0
<i>sect163r2</i>	26	0.0
<i>secp224r1</i>	3	0.0
<i>secp192r1</i>	1	0.0

La courbe *secp256r1* se sert de paramètres d'une taille de 256 bits, ce qui offre une sécurité plus forte encore que les clés de 2048 bits pour RSA, comme le soulignent les auteurs de [13].

Ainsi, on observe non seulement que les paramètres utilisés pour ECDHE offrent une meilleure sécurité que ceux de DHE avec une taille largement inférieure, mais qu'en plus la majorité des serveurs utilisant DHE fait appel à des paramètres de 1024 bits, ce qui est insuffisant en termes de sécurité. Au contraire, les serveurs utilisant ECDHE font appel pour la plupart à des paramètres qui offrent une sécurité bien supérieure.

Outre la taille des clés utilisées, les auteurs de [13] ont aussi mesuré le temps d'exécution nécessaire au *Handshake Protocol* de TLS ainsi qu'à la vérification des certificats pour les diverses *cipher suites* disponibles dans le protocole. Les deux figures suivantes, tirées de [13], montrent le nombre moyen de requêtes pouvant être traitées par secondes par un serveur TLS pour différentes *cipher suites*, et respectivement le temps d'exécution pris en milli-secondes côté client pour une

connexion à un serveur TLS sur le navigateur Chrome, sous différents OS :

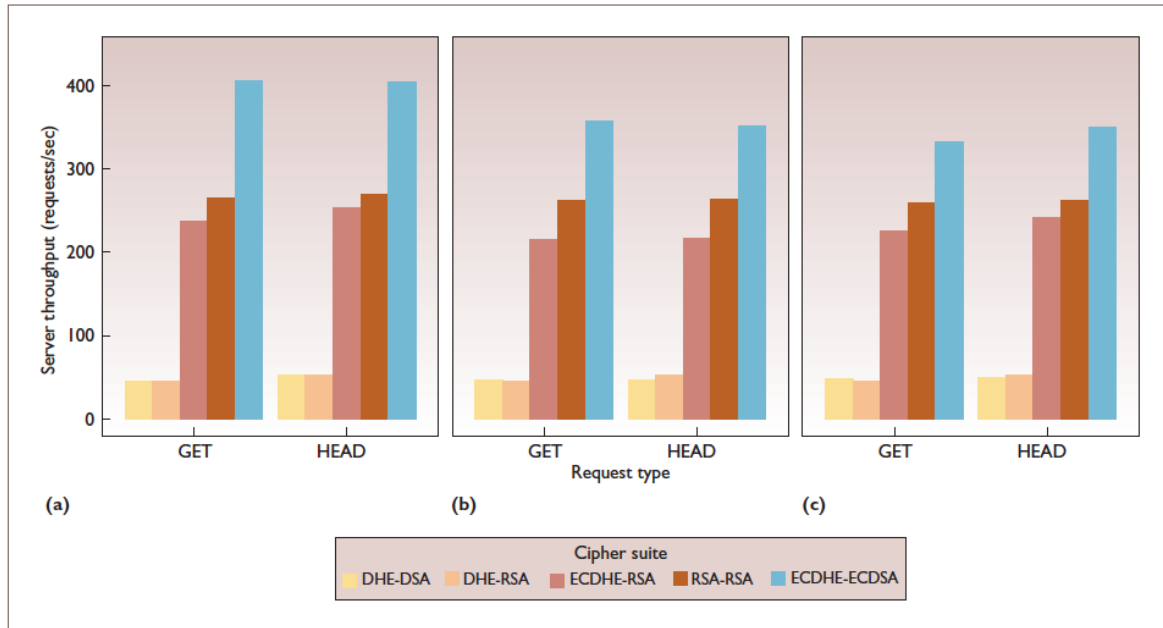


Figure 1. Server throughput of different configurations under synthetic traffic for an (a) simple page, (b) complex page, and (c) multidomain page.

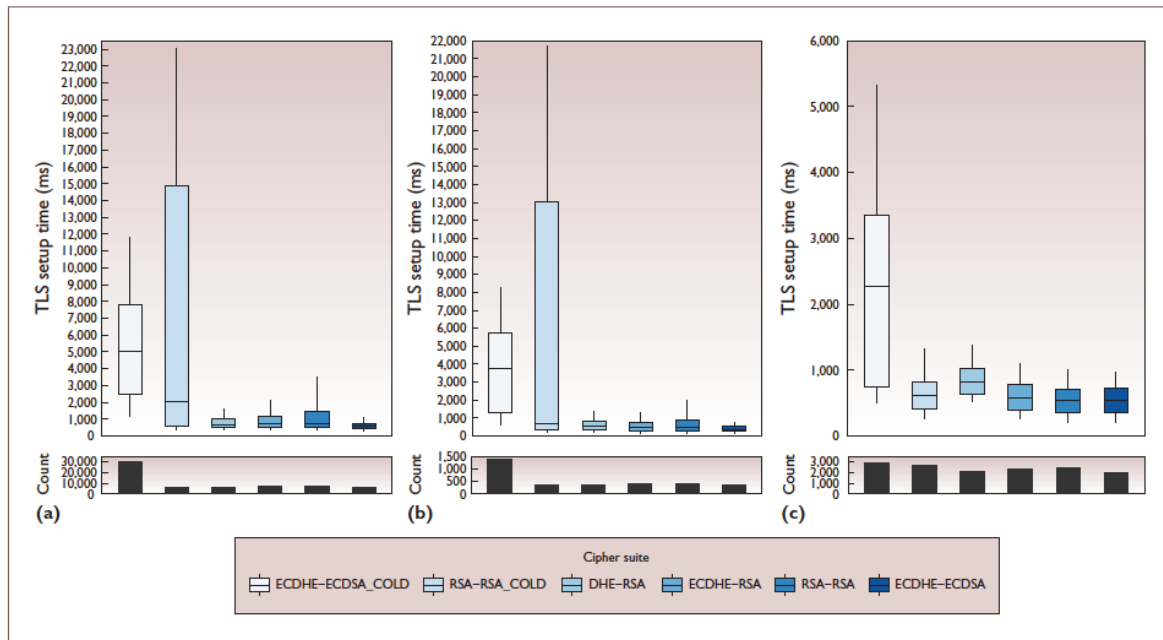


Figure 2. Comparison of TLS setup times in Chrome browsers on (a) Windows, (b) OS X, and (c) Android. The box plots show the 10th, 25th, 50th, 75th, and 90th percentiles of measured TLS setup times for each cipher suite. The corresponding bar charts show the number of unique clients that successfully completed each test.

On remarque dans la première image que DHE est clairement l'algorithme le plus lent à s'exécuter pour la négociation des clés entre serveur et client. ECDHE couplé avec RSA pour l'authentification est presque aussi efficace que RSA-RSA, et ECDHE-ECDSA est même plus rapide encore.

Dans la seconde image, les suites cryptographiques dites « *cold* » (ECDHE-ECDSA_COLD et RSA-RSA_COLD) correspondent à l'utilisation des *cipher suites* lors d'une connexion avec un cache vide, et donc avec une vérification des certificats côté client. Ceci explique les temps plus longs pour ces deux cas. On observe que les temps les plus courts sont obtenus lorsque les courbes elliptiques sont utilisées pour la négociation des clés, mais aussi pour l'authentification. Cela confirme donc l'idée que l'utilisation de la cryptographie à courbes elliptiques, en particulier sur l'algorithme de Diffie-Hellman, présente l'avantage d'être plus rapide que celle de l'algorithme simple. Ainsi, en plus d'offrir la *perfect forward secrecy*, ECDHE est aussi l'algorithme de négociation des clés le plus efficace dans TLS.

3.4 Considérations supplémentaires :

Dans [16], l'auteur décrit l'importance de la *perfect forward secrecy* sur internet et les moyens de rendre son support possible dans TLS, mais il souligne aussi le fait que cette propriété ne doit pas non plus être vue comme un gage de sécurité absolue pour des échanges cryptés. En effet, bien que la PFS offre sur le long terme une assurance que les clés de session utilisées dans un échange ne pourront être retrouvées par un attaquant, elle n'assure pas que l'échange ne sera pas compromis autrement, lorsqu'il a lieu par exemple, si un attaquant est capable de casser l'algorithme de cryptographie symétrique utilisé lors d'une communication (en particulier si cet algorithme offre une sécurité faible, comme cela est le cas, entre autres, de RC4 ou DES).

De plus, l'auteur explique que si un algorithme comme RSA est utilisé pour l'authentification lors d'une session TLS et qu'un attaquant vient à entrer en possession de la clé privée du serveur, les connexions ultérieures ne seront plus sûres, même si la *cipher suite* utilisée dans l'échange supporte la PFS. En effet, si un attaquant entre en possession de la clé privée du serveur dans un échange, il peut alors monter une attaque *Man in the Middle* : il sera capable d'intercepter et de modifier les paramètres de DH envoyés au client lors du *Handshake Protocol*, puis de les signer avec la clé volée. Le client confirmera alors la signature sur les paramètres comme authentique et pensera communiquer avec son correspondant légitime, alors qu'il aura en réalité affaire à une entité mal intentionnée.

Ainsi, bien que la PFS soit une propriété devenue à ce jour essentielle dans les échanges sécurisés sur internet, il ne faut pas non plus négliger d'autres facteurs lors de la mise en place et de l'implémentation du protocole TLS sur des serveurs et dans des applications côté client, comme la fiabilité des algorithmes de cryptographie symétrique utilisés, l'importance d'avoir accès à des certificats à jour pour l'authentification, l'utilisation de générateurs pseudo-aléatoires sûrs pour la génération des secrets, etc... Une description des attaques et vulnérabilités connues à ce jour pour TLS est donnée dans [2] et [3], ainsi que des recommandations pour la configuration de TLS afin de contrer ces dernières.

4. Le déploiement de la PFS dans TLS sur internet :

À partir de 2011, plusieurs sites importants, tels que Google ([17]), Facebook, Twitter ([18]) ou encore GitHub ([19]) ont annoncé qu'ils allaient adopter la *perfect forward secrecy*. Depuis, un nombre croissant de sites a suivi leur exemple, et la PFS devient petit à petit la norme sur le web.

Deux études publiées en 2014, [13] et [16], ont permis d'évaluer le taux de déploiement de la PFS dans TLS sur internet. Dans [13], les auteurs ont développé un programme afin de scanner

les serveurs appartenant au million de sites les plus visités sur internet (selon le classement de Alexa.com). Ils ont effectué leurs mesures entre le 13 et le 27 septembre 2013. Leur scanner leur a permis, entre autres, de déterminer les versions de TLS acceptées par les serveurs analysés, ainsi que les suites cryptographiques utilisées dans le protocole. Ainsi, les auteurs ont pu se connecter avec succès sur 473'802 serveurs TLS (la plupart des échecs de connexion étant dus au fait que les serveurs concernés n'utilisaient pas TLS), et les résultats obtenus sont les suivants : sur 473'802 serveurs, 99.9 % (soit 473'688) supportaient RSA comme algorithme d'échange de clés, 59.8 % (283'647) supportaient DHE, 17.9 % (85'070) supportaient ECDHE, et moins d'un pourcent supportait ECDH. Au total, 74.5 % des serveurs (soit 353'209) supportaient donc la PFS, en utilisant DHE ou ECDHE. De plus, 81 % de ces serveurs donnaient la priorité aux *cipher suites* offrant la PFS.

La tendance générale était donc clairement à l'utilisation de RSA en 2013, mais pratiquement trois quarts des serveurs offraient tout de même la possibilité de faire appel à des *cipher suites* offrant la PFS. Malheureusement, comme expliqué plus haut, de nombreux serveurs utilisant DHE faisaient appel à des paramètres d'une longueur de moins de 2048 bits, offrant donc une sécurité restreinte pour l'échange de clés.

Dans [16], l'auteur a lui aussi analysé les serveurs d'un nombre important de sites afin de déterminer les versions de TLS supportées par ces derniers et les *cipher suites* utilisées : en avril 2014, il a scanné les 20'000 sites les plus visités selon le site *Alexa.com*, ainsi que les sites appartenant au domaine *.jp* faisant partie du million de sites les plus visités selon le même classement. Il n'est parvenu à se connecter via le port 443 (c'est-à-dire avec une connexion TLS) qu'à 6'835 des sites du classement *Alexa.com* et 5'668 sites du domaine *.jp*. Il a alors compté le nombre de serveurs supportant chaque suite cryptographique offrant la PFS, et il a obtenu les résultats suivants (figure tirée de [16]) :

(Total)	A20K 6835	.jp 5668
EXP-EDH-RSA-DES-CBC-SHA	365	1644
EDH-RSA-DES-CBC-SHA	640	2803
DHE-RSA-AES256-SHA	2515	3879
DHE-RSA-CAMELLIA256-SHA	1394	1254
DHE-RSA-AES128-SHA256	922	557
DHE-RSA-AES128-SHA	2614	3906
DHE-RSA-CAMELLIA128-SHA	1391	1254
DHE-RSA-SEED-SHA	659	979
EDH-RSA-DES-CBC3-SHA	2413	3897
DHE-RSA-AES256-GCM-SHA384	923	556
DHE-RSA-AES128-GCM-SHA256	921	555
ECDHE-RSA-AES256-SHA	2081	373
ECDHE-RSA-AES128-SHA	2079	374
ECDHE-RSA-AES256-GCM-SHA384	1573	105
ECDHE-RSA-AES128-GCM-SHA256	1614	109
ECDHE-RSA-DES-CBC3-SHA	1686	126
ECDHE-RSA-RC4-SHA	1448	117

On observe que l'algorithme d'échange de clés avec PFS le plus supporté est DHE, comme c'était le cas dans [13], dont les mesures avaient été effectuées quelques mois auparavant. On remarque toutefois que beaucoup de sites dans la liste offrent aussi la possibilité d'utiliser ECDHE. Tout comme dans [13], l'auteur de [16] a remarqué que la plupart des serveurs se servant de DHE faisaient encore appel à des paramètres d'une longueur de moins de 2048 bits pour l'algorithme : il n'y a donc eu que peu d'évolution à ce niveau dans les mois qui ont séparé les deux études.

En janvier 2014, Julien Vehent, spécialiste de la sécurité chez Mozilla, a lui aussi scanné les sites appartenant au *top 1 million* de Alexa.com. Il a rapporté ses résultats dans [20] : tout comme dans les publications mentionnées ci-dessus, il est arrivé à la conclusion que 75 % des serveurs sur lesquels il a pu se connecter via TLS offraient la PFS. Il a aussi remarqué que la plupart d'entre eux utilisaient des paramètres trop faibles pour DHE, ce qu'il explique par le fait que dans Apache 2.4.6 et les versions précédentes, la taille des paramètres pour DHE est automatiquement réglée à 1024 bits et n'est pas paramétrable. De plus, il note que Java 6 ne supporte pas les clés plus grandes que cette longueur. Une observation supplémentaire faite dans cet article indique que 60 % des serveurs offrent non seulement la PFS mais la privilégient lors du choix de la *cipher suite* utilisée dans TLS, ce qui corrobore les chiffres publiés dans [13].


Un autre article, [21], publié en septembre 2014 sur le *security blog* de la compagnie RedHat, confirme une fois de plus les résultats donnés plus haut concernant le support de la *perfect forward secrecy* par les serveurs des sites appartenant au *top 1 million* de Alexa.com. L'auteur de ce dernier note de plus que le nombre de serveurs négociant des suites cryptographiques supportant la PFS est en croissance constante.

L'entreprise de sécurité informatique Qualys offre sur son site *SSL Labs* ([22]) plusieurs services permettant d'estimer l'état actuel ainsi que la qualité des implémentations de SSL/TLS sur internet. L'un de ces services, *SSL Pulse* [23], offre des statistiques régulièrement mises à jour concernant l'implémentation de TLS sur les serveurs des 200'000 sites les plus visités du web. L'une de ces statistiques mesure le taux d'adoption de la PFS : en août 2015, 30.4 % des serveurs ne supporte pas la PFS, soit 1.6 % de moins que le mois précédent. 34 % des serveurs supporte certaines *cipher suites* offrant la PFS, 17.5 % supporte la PFS avec les navigateurs web modernes (c'est-à-dire lorsqu'un navigateur web moderne, tel que Firefox ou Chrome dans leurs versions récentes, est utilisé pour se connecter sur le site) et 18.1 % la supporte avec la plupart des navigateurs web. On observe ainsi que l'adoption de la PFS est à ce jour encore loin d'être totale sur le web, mais qu'elle progresse néanmoins petit à petit.

5. Recommandations pratiques :

Pour l'implémentation et la configuration d'un serveur TLS, de nombreuses précautions sont à observer afin d'offrir aux utilisateurs la meilleure sécurité possible à ce jour avec le protocole. La fondation Mozilla, un organisme qui promeut la neutralité et la sécurité sur internet, offre de nombreuses recommandations à ce sujet sur son Wiki : [24]. Une section y est en particulier dédiée au support de la *perfect forward secrecy* : les algorithmes la supportant y sont décrits, et des indications sont données quant aux tailles des paramètres à utiliser ainsi que d'autres détails de configuration.

Côté client, la plupart des navigateurs web modernes sont aujourd'hui compatibles avec les *cipher suites* qui offrent la PFS, et la majorité d'entre eux utilise même par défaut et en priorité de telles suites cryptographiques lors de connexions TLS. Une analyse des navigateurs Firefox, Chrome et Safari avec l'outil de test de browser trouvé sur *SSL Labs* ([25]) donne les résultats suivants quant aux *cipher suites* utilisées par ces derniers, par ordre de préférence (avec les paramètres par défaut de ces navigateurs activés) :




Cipher Suites (in order of preference)

TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)	Forward Secrecy	128
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)	Forward Secrecy	128
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)	Forward Secrecy	256
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)	Forward Secrecy	128
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)	Forward Secrecy	128
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)	Forward Secrecy	256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0xc33)	Forward Secrecy	128
TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0xc39)	Forward Secrecy	256
TLS_RSA_WITH_AES_128_CBC_SHA (0x2f)		128
TLS_RSA_WITH_AES_256_CBC_SHA (0x35)		256
TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xa)		112

(1) When a browser supports SSL 2, its SSL 2-only suites are shown only on the very first connection to this site. To see the suites, close all browser windows, then open this exact page directly. Don't refresh.

Liste des cipher suites supportées par Firefox, par ordre de préférence.

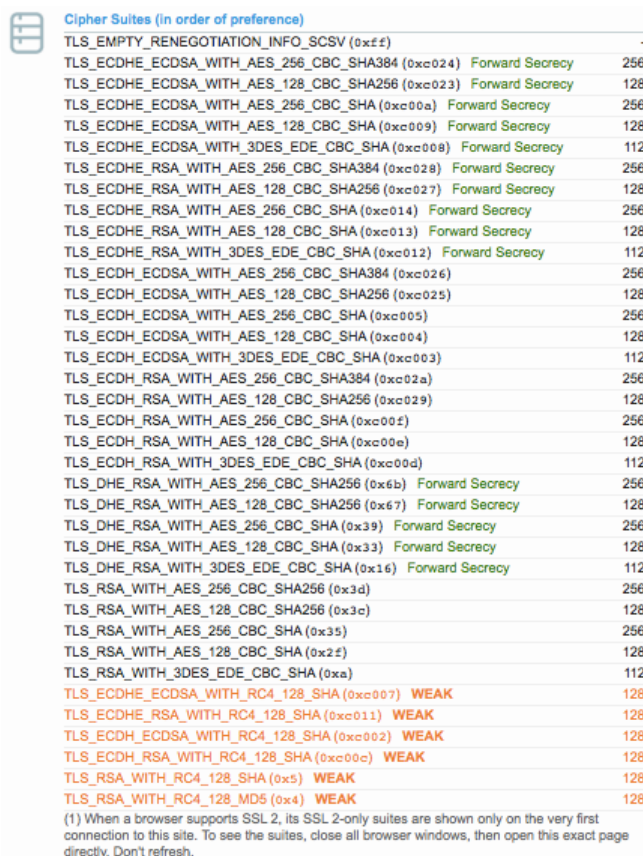


Cipher Suites (in order of preference)

TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)	Forward Secrecy	128
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)	Forward Secrecy	128
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x9e)	Forward Secrecy	128
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc14)	Forward Secrecy	256
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc13)	Forward Secrecy	256
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcc15)	Forward Secrecy	256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)	Forward Secrecy	256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)	Forward Secrecy	256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0xc39)	Forward Secrecy	256
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)	Forward Secrecy	128
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)	Forward Secrecy	128
TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0xc33)	Forward Secrecy	128
TLS_RSA_WITH_AES_128_GCM_SHA256 (0x9c)		128
TLS_RSA_WITH_AES_256_CBC_SHA (0x35)		256
TLS_RSA_WITH_AES_128_CBC_SHA (0x2f)		128
TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xa)		112
TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0xff)		-

(1) When a browser supports SSL 2, its SSL 2-only suites are shown only on the very first connection to this site. To see the suites, close all browser windows, then open this exact page directly. Don't refresh.

Liste des cipher suites supportées par Chrome.



Cipher Suites (in order of preference)

TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0xc0000000)	-
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc0024)	Forward Secrecy 256
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc0023)	Forward Secrecy 128
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc002a)	Forward Secrecy 256
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc0009)	Forward Secrecy 128
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc0008)	Forward Secrecy 112
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc0028)	Forward Secrecy 256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc0027)	Forward Secrecy 128
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc0014)	Forward Secrecy 256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc0013)	Forward Secrecy 128
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc0012)	Forward Secrecy 112
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384 (0xc0026)	256
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256 (0xc0025)	128
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc0005)	256
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc0004)	128
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc0003)	112
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384 (0xc002a)	256
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256 (0xc0029)	128
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc000f)	256
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc000e)	128
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA (0xc000d)	112
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0xc06b)	Forward Secrecy 256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0xc067)	Forward Secrecy 128
TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0xc039)	Forward Secrecy 256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0xc033)	Forward Secrecy 128
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc016)	Forward Secrecy 112
TLS_RSA_WITH_AES_256_CBC_SHA256 (0xc03d)	256
TLS_RSA_WITH_AES_128_CBC_SHA256 (0xc03c)	128
TLS_RSA_WITH_AES_256_CBC_SHA (0xc035)	256
TLS_RSA_WITH_AES_128_CBC_SHA (0xc02f)	128
TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xc00a)	112
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc0007)	WEAK 128
TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc0011)	WEAK 128
TLS_ECDH_ECDSA_WITH_RC4_128_SHA (0xc0002)	WEAK 128
TLS_ECDH_RSA_WITH_RC4_128_SHA (0xc000c)	WEAK 128
TLS_RSA_WITH_RC4_128_SHA (0xc005)	WEAK 128
TLS_RSA_WITH_RC4_128_MD5 (0xc004)	WEAK 128

(1) When a browser supports SSL 2, its SSL 2-only suites are shown only on the very first connection to this site. To see the suites, close all browser windows, then open this exact page directly. Don't refresh.

Liste des cipher suites supportées par Safari.

Bien que les *cipher suites* respectant la PFS soient privilégiées, ces navigateurs web permettent aussi l'utilisation d'autres suites cryptographiques. Ainsi, lors d'une connexion sur un serveur qui n'offre pas de support pour les suites avec la PFS, d'autres seront utilisées, et la propriété ne sera pas respectée pour l'échange de données. Il est donc possible de modifier les paramètres de ces *browsers* afin d'y remédier, en désactivant les *cipher suites* sans PFS. Nous allons décrire ici la façon de procéder pour modifier ces paramètres. Il est important toutefois de noter que de telles modifications sur les *browsers* peuvent entraîner des incompatibilités avec certains serveurs web et donc empêcher la connexion sur ces derniers. C'est d'ailleurs pour cette raison que la plupart des navigateurs, bien qu'ils privilégient la PFS, permettent l'usage de *cipher suites* ne respectant pas cette propriété, lorsque leur utilisation est absolument nécessaire, et pour des raisons de compatibilité avec certains serveurs.

5.1 Configurer Firefox [26] :

Firefox offre une grande liberté à ses utilisateurs pour la configuration de nombreux paramètres. L'accès à ces derniers est très simple sur ce *browser* : il suffit de taper « *about:config* » dans la barre d'adresse. Après l'affichage d'un message d'avertissement, une page de configuration est ouverte, avec une longue liste d'options auxquelles sont associées un statut, un type et une valeur. Pour accéder facilement aux options concernant les *cipher suites* activées dans le navigateur, le plus simple est de taper dans la barre de recherche l'entrée : « *security.ssl3* ». La liste de toutes les suites cryptographiques supportées par Firefox s'affiche alors. Pour désactiver l'utilisation d'une *cipher suite*, il suffit de double cliquer dessus afin que sa valeur soit mise à *false*. Pour s'assurer un support total de la PFS (et ce, peut-être, au risque de ne plus pouvoir se connecter à certains sites

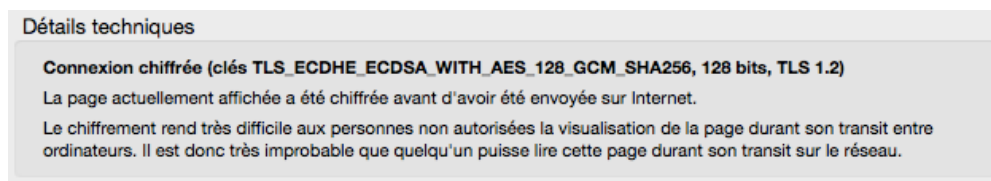
utilisant TLS avec les *cipher suites* désactivées), il faut désactiver toutes les *cipher suites* dont le nom commence par « *rsa* » (le premier nom d'algorithme pour chaque suite cryptographique représente celui qui est utilisé pour l'échange de clés).

Rechercher : security.ssl3

Nom de l'option	Statut	Type	Valeur
security.ssl3.dhe_rsa_aes_128_sha	par défaut	booléen	true
security.ssl3.dhe_rsa_aes_256_sha	par défaut	booléen	true
security.ssl3.ecdhe_ecdsa_aes_128_gcm_sha256	par défaut	booléen	true
security.ssl3.ecdhe_ecdsa_aes_128_sha	par défaut	booléen	true
security.ssl3.ecdhe_ecdsa_aes_256_sha	par défaut	booléen	true
security.ssl3.ecdhe_ecdsa_rc4_128_sha	par défaut	booléen	true
security.ssl3.ecdhe_rsa_aes_128_gcm_sha256	par défaut	booléen	true
security.ssl3.ecdhe_rsa_aes_128_sha	par défaut	booléen	true
security.ssl3.ecdhe_rsa_aes_256_sha	par défaut	booléen	true
security.ssl3.ecdhe_rsa_rc4_128_sha	par défaut	booléen	true
security.ssl3.rsa_aes_128_sha	par défaut	booléen	true
security.ssl3.rsa_aes_256_sha	par défaut	booléen	true
security.ssl3.rsa_des_ede3_sha	par défaut	booléen	true
security.ssl3.rsa_rc4_128_md5	par défaut	booléen	true
security.ssl3.rsa_rc4_128_sha	par défaut	booléen	true

La liste des cipher suites supportées par Firefox, dans l'outil de configuration.

Une connexion à un serveur via HTTPS est représentée dans Firefox par un cadenas dans la barre d'adresse. Le navigateur offre la possibilité à l'utilisateur de voir certaines données et paramètres de sécurité utilisés pour la connexion TLS associée, tels que les certificats du serveur par exemple. Il est aussi possible de voir la *cipher suite* utilisée pour les échanges via TLS. Pour ce faire, il suffit de cliquer sur le cadenas dans la barre d'adresse. Une petite boîte de dialogue s'ouvre alors. En cliquant sur le bouton « Plus d'informations » en bas de cette dernière, on accède à une fenêtre dans laquelle est indiquée, dans l'encadré « Détails techniques », la *cipher suite* négociée avec le serveur lors du *Handshake Protocol*.



Exemple de cipher suite utilisée lors d'une connexion à Gmail, comme affichée par Firefox lorsque le bouton « plus d'informations » est cliqué.

5.2 Configurer Chrome :

La configuration des *cipher suites* dans Chrome n'est pas aussi simple que sur Firefox. En effet, Chrome n'offre pas d'outil de configuration pour modifier un tel paramètre. Il est toutefois possible de désactiver certaines suites cryptographiques, comme cela est le cas dans Firefox, en les « blacklistant ». Pour ce faire, il faut ajouter des arguments de ligne de commande à l'exécutable du navigateur. La méthode diffère selon les systèmes d'exploitation : sous Windows, il faut effectuer un clic droit sur le raccourci utilisé pour lancer Chrome, sélectionner « Propriétés », puis ajouter à la fin de la zone de texte « cible » la commande : « `--cipher-suite-blacklist=0x9c,0x35,0x2f,0xa,0xff` ».

Les valeurs hexadécimales à la fin de la commande représentent les *cipher suites* à désactiver. Les valeurs associées à chaque suite cryptographique sont affichées dans la figure concernant les suites supportées par Chrome, plus haut dans ce document. Elles correspondent au *cipher suites* utilisant RSA pour l'échange de clés.

Pour effectuer la même opération sous MacOS X, il faut lancer le terminal (/Applications/Utilities/Terminal.app), puis entrer la commande : « /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --cipher-suite-blacklist=0x9c,0x35,0x2f,0xa,0xff ». Ceci lancera l'application Chrome avec les suites cryptographiques correspondantes désactivées.

Enfin, sous Linux, il suffit de taper dans la console la commande : « google-chrome - cipher-suite-blacklist=0x9c,0x35,0x2f,0xa,0xff ».

Tout comme dans Firefox, il est possible avec Chrome de voir quelle *cipher suite* est utilisée lors d'une connexion HTTPS sur un site internet. Pour ce faire, il suffit de cliquer sur le cadenas qui s'affiche dans la barre d'adresse du navigateur, puis de cliquer sur l'onglet « connexion ». Les différents algorithmes utilisés pour l'authentification, l'échange de clés et le chiffrement des données sont alors affichés.

5.3 Autres navigateurs :

Safari n'offre pas la possibilité à ses utilisateurs de configurer les *cipher suites* pouvant être utilisées lors d'échanges via TLS. Ainsi, l'utilisateur est dépendant des choix faits par les développeurs de l'application quand aux suites cryptographiques utilisées et privilégiées par le programme. Il n'est donc pas possible de s'assurer un support « absolu » de la PFS avec ce *browser*.

Il est possible de désactiver l'utilisation de certaines cipher suites dans le navigateur Opera, exactement de la même façon que pour Chrome, avec la commande « --cipher-suite-blacklist=... », ajoutée à la fin du chemin de l'exécutable.

6. Conclusion :

La *perfect forward secrecy* est devenue à ce jour une propriété indispensable pour la sécurité sur le long terme des échanges sur internet. Elle permet de s'assurer que même si des communications cryptées sont observées et enregistrées par un attaquant, leur confidentialité ne pourra pas être compromise plus tard par la découverte d'une des clés secrètes des participants de l'échange. Ainsi, elle rend les communications sécurisées résistantes à l'épreuve du temps et de l'évolution des technologies utilisées pour casser les algorithmes de cryptographie modernes.

Bien qu'introduite pour la première fois dans le monde de la sécurité informatique il y a plusieurs décennies par Whitfield Diffie, Paul van Oorschot et Michael James Wiener, la PFS est encore loin d'avoir été totalement adoptée sur internet, et son expansion et sa promotion par divers organismes défendant la neutralité et la sécurité du web ne sont que très récents : ce n'est que depuis une petite dizaine d'années que l'on observe véritablement sur internet l'adoption de cette propriété par de nombreux sites et serveurs.

SSL/TLS étant aujourd'hui le protocole de sécurité le plus important et le plus utilisé sur internet, c'est principalement à travers ce dernier que se propage la PFS sur internet. Il semble

normal que TLS soit l'un des protocoles devant offrir en priorité une telle propriété, puisqu'il joue un rôle central dans la confidentialité des échanges sur le réseau. Il doit donc pouvoir assurer la sécurité sur le long terme des communications chiffrées. Comme expliqué dans ce document, il est nécessaire de respecter de nombreux paramètres et d'implémenter TLS selon une configuration bien précise pour assurer le support de la PFS par le protocole.

On observe aujourd'hui que les implémentations de TLS offrant la PFS dans les serveurs sur internet ne sont pas majoritaires, et que de nombreux sites permettent encore l'utilisation de *cipher suites* sans support pour cette propriété. Certains n'utilisent même aucune suite cryptographique assurant la PFS. Il est toutefois encourageant de voir que le nombre de serveurs offrant la PFS est en croissance constante par rapport à ceux qui ne la supportent pas, et que son adoption devient petit à petit la norme sur internet. Aujourd'hui, de plus en plus d'organismes jouant un rôle central dans la sécurité des réseaux participent activement à la promotion de la PFS, et les outils permettant aux développeurs de la mettre en place sont de plus en plus nombreux. Il est aussi intéressant de noter que la majorité des navigateurs web privilégient par défaut les *cipher suites* offrant la PFS dans leur implémentation et permettent même, pour certains, de choisir lesquelles seront disponibles lors d'échanges via TLS. Ceci souligne l'engagement de la communauté des développeurs dans la sécurité des échanges sur internet et présage une évolution positive de l'utilisation de TLS sur le web.

À la lumière d'événements récents tels que la faille *Heartbleed* du protocole TLS, ou encore les scandales des écoutes menées par des agences de sécurité nationale sur les données de millions de personnes dans la monde, la PFS semble être devenue une propriété indispensable pour la sécurité des échanges sur internet. Il semble donc crucial de sensibiliser les utilisateurs à son importance afin de faire progresser son adoption sur internet, dans le but de permettre un jour son support pour toutes les communications sécurisées transitant par le réseau et d'en faire une propriété universelle des échanges chiffrés.

Bibliographie

- [1] RFC 2246 : <https://www.ietf.org/rfc/rfc2246.txt>.
- [2] O. Levillain, B. Gourdin, and H. Debar, "TLS Record Protocol: Security Analysis and Defense-in-depth Countermeasures for HTTPS", 2015, pp. 225–236.
- [3] M. Atighetchi, N. Soule, P. Pal, J. Loyall, A. Sinclair, and R. Grant, "Safe configuration of TLS connections", 2013, pp. 415–422.
- [4] "Forward secrecy," *Wikipedia, the free encyclopedia*. 23-Jun-2015 : https://en.wikipedia.org/w/index.php?title=Forward_secrecy&oldid=668218881.
- [5] E. Solana, "Cryptographie et sécurité, Théorie et pratique." 2015.
- [6] "Perfect Forward Secrecy - An Introduction." : <https://scotthelme.co.uk/perfect-forward-secrecy/>.
- [7] "Why the Web Needs Perfect Forward Secrecy More Than Ever," *Electronic Frontier Foundation* : <https://www.eff.org/deeplinks/2014/04/why-web-needs-perfect-forward-secrecy>.
- [8] W. Stallings and L. Brown, *Computer security: principles and practice*, 3. ed., global ed. Boston, Pearson, 2015, pp. 720-727.
- [9] RFC 4346 : <https://www.ietf.org/rfc/rfc4346.txt>.
- [10] RFC 5246 : <https://www.ietf.org/rfc/rfc5246.txt>.
- [11] "SSL/TLS & Perfect Forward Secrecy | Vincent Bernat." :

- <http://vincent.bernat.im/fr/blog/2011-ssl-perfect-forward-secrecy.html>.
- [12] “SSL: Foundation for Web Security - The Internet Protocol Journal - Volume 1, No. 1,” *Cisco* : http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_1-1/ssl.html.
 - [13] L.-S. Huang, S. Adhikarla, D. Boneh, and C. Jackson, “An Experimental Study of TLS Forward Secrecy Deployments,” *IEEE Internet Comput.*, vol. 18, no. 6, pp. 43–51, Nov. 2014.
 - [14] Ç. K. Koç, Ed., *Cryptographic engineering*. New York, NY, USA: Springer, 2009.
 - [15] S. Blake-Wilson, B. Moeller, V. Gupta, C. Hawk, and N. Bolyard, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS).” : <https://tools.ietf.org/html/rfc4492>.
 - [16] Y. Suga, “SSL/TLS Servers Status Survey about Enabling Forward Secrecy,” 2014, pp. 501–505.
 - [17] “Protecting data for the long term with forward secrecy,” *Google Online Security Blog*. : <http://googleonlinesecurity.blogspot.com/2011/11/protecting-data-for-long-term-with.html>.
 - [18] “Forward Secrecy at Twitter,” *Twitter Blogs* : <https://blog.twitter.com/2013/forward-secrecy-at-twitter>.
 - [19] dbussink, “Introducing Forward Secrecy and Authenticated Encryption Ciphers,” *GitHub* : <https://github.com/blog/1727-introducing-forward-secrecy-and-authenticated-encryption-ciphers>.
 - [20] J. Vehent, “SSL/TLS Analysis of the Internet’s top 1,000,000 websites,” 11-Jan-2014 : https://jve.linuxwall.info/blog/index.php?post/TLS_Survey.
 - [21] H. Kario, “TLS Landscape,” *Red Hat Security* : <https://securityblog.redhat.com/2014/09/10/tls-landscape>.
 - [22] “Qualys SSL Labs.” : <https://www.ssllabs.com/>.
 - [23] “Trustworthy Internet Movement - SSL Pulse.” : <https://www.trustworthyinternet.org/ssl-pulse/>.
 - [24] “Security/Server Side TLS - MozillaWiki.” : https://wiki.mozilla.org/Security/Server_Side_TLS.
 - [25] “Qualys SSL Labs - Projects / SSL Client Test.” : <https://www.ssllabs.com/ssltest/viewMyClient.html>.
 - [26] “About:config - MozillaZine Knowledge Base.” : <http://kb.mozillazine.org/About:config>.