SSL/TLS, 3 ans plus tard

Olivier Levillain
olivier.levillain@ssi.gouv.fr

ANSSI

Résumé Depuis quelques années, l'actualité autour de SSL/TLS s'est accélérée. De nombreuses failles ont été mises au jour, qu'il s'agisse de faiblesses conceptuelles concernant la phase de négociation, de vulnérabilités cryptographiques affectant la protection des flux ou d'erreurs d'implémentation.

Un premier état des lieux avait été dressé en 2012. Cet article est composé de trois parties : la première se veut être une actualisation de la présentation de 2012 ; la seconde décrit les nombreuses failles d'implémentation qui ont touché toutes les piles SSL/TLS majeures en 2014 ; enfin, la dernière partie présente des pistes pour l'avenir du protocole, avec des éléments d'analyse de TLS 1.3, en cours de définition à l'IETF.

Mots-clés: SSL/TLS, implémentations, HTTPS

Introduction

SSL (*Secure Sockets Layer*) et son successeur TLS (*Transport Layer Security*) sont des protocoles dont l'objectif est de fournir un certain nombre de services pour sécuriser un canal de communication : authentification unilatérale ou mutuelle, confidentialité, intégrité et non-rejeu des données échangées de bout en bout.

Cette couche de sécurité peut être appliquée à tout type de canal de communication entre deux parties garantissant la transmission des données de façon ordonnée. En pratique, SSL/TLS est surtout utilisé sur la couche transport TCP, afin de proposer des versions sécurisées de protocoles existants (par exemple HTTPS = HTTP + SSL). Il existe une variante de TLS reposant sur une couche de transport non fiable, DTLS (*Datagram Transport Layer Security*), qui est peu utilisée en pratique.

En 2012, un article présenté au SSTIC avait dressé un état des lieux de SSL/TLS [35]. Depuis cette date, l'actualité concernant SSL/TLS a continué de faire les gros titres de la presse spécialisée : CRIME, Lucky13, Heartbleed ou POODLE, pour n'en citer que quelques-uns. Ce document a pour objectif de proposer une mise à jour du panorama sur SSL/TLS.

La section 1 de ce document présente rapidement l'historique et le fonctionnement de SSL/TLS. Les section 2, 3 et 4 résument et enrichissent l'état des lieux réalisé en 2012. La section 5 décrit les failles d'implémentations qui ont affecté les principales implémentations de SSL/TLS en 2014. Enfin, la section 6 présente des réflexion quant aux améliorations possibles, et ce que TLS 1.3, nouvelle version du standard en cours de définition, peut apporter.

1 SSL/TLS : un peu de contexte

1.1 Historique

SSL (*Secure Sockets Layer*) est un protocole mis au point par Netscape à partir de 1994 pour permettre l'établissement d'une connexion sécurisée (chiffrée, intègre et authentifiée). La première version publiée est la version 2 [28], rendue disponible en 1995. SSLv2 fut rapidement suivi d'une version 3 [22], qui corrige des failles conceptuelles importantes. Bien qu'il existe un mode de fonctionnement de compatibilité, les messages de la version 3 diffèrent de ceux de la version 2.

En 2001, SSL a fait l'objet d'une standardisation par l'IETF (*Internet Engineering Task Force*) et a été renommé TLS (*Transport Layer Security*). Contrairement au passage de SSLv2 à SSLv3, TLS n'a pas été l'objet de changements structurels.

Depuis 2001, TLS a connu quelques évolutions. Dès 2003, un cadre permettant des extensions dans le protocole TLS a été décrit [10]. Ce cadre a été réactualisé en 2006 et 2011 [11,18]. Ces extensions sont aujourd'hui essentielles pour permettre de faire évoluer le standard de façon souple, mais requièrent l'abandon de la compatibilité avec SSLv2 et SSLv3. La version la plus récente du protocole est actuellement TLS 1.2, publiée en 2008 [16].

1.2 Détails d'une connexion TLS classique

Afin de permettre l'établissement d'un canal de communication chiffré et intègre, les deux parties doivent s'entendre sur les algorithmes et les clés à utiliser. Dans cette étape de négociation, plusieurs messages sont échangés. Un exemple complet est donné à la figure 1. On suppose pour l'exemple que la suite cryptographique négociée est `TLS_RSA_WITH_AES_128_CBC_SHA`.

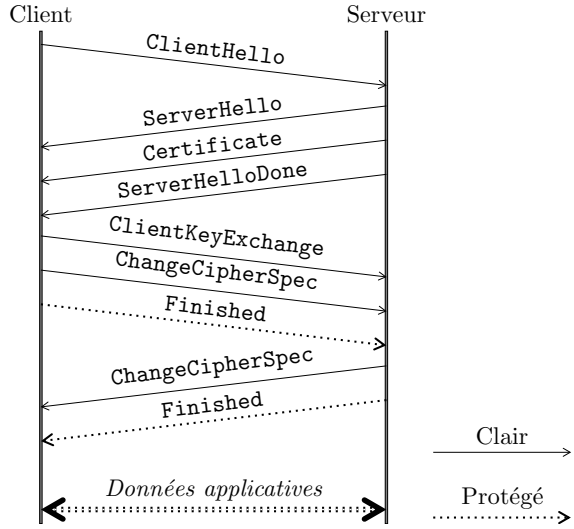


Figure 1. Exemple de négociation TLS avec la suite `TLS_RSA_WITH_AES_128_CBC_SHA`.

`ClientHello` La négociation commence avec l'envoi par le client d'un message `ClientHello`. Dans ce message, le client propose un ensemble de suites cryptographiques qu'il est capable de mettre en œuvre. Chacune de ces suites cryptographiques décrivent les mécanismes cryptographiques qui seront utilisés pour les fonctions suivantes :

— l'échange de clés ;
— l'authentification du serveur [1] ;
— la protection des données applicatives, en confidentialité et en intégrité.

Ce message contient d'autres paramètres qui doivent être négociés : la version du standard utilisée (SSLv3, TLS 1.0, TLS 1.1 ou TLS 1.2) et le mécanisme de compression éventuellement appliqué sur les données applicatives. Enfin, il comporte un champ `client_random`, un aléa fourni par le client qui sera utilisé pour la dérivation des clés.

Réponse du serveur Lorsque le serveur reçoit le `ClientHello`, deux cas peuvent se produire :

— aucune des propositions du client n'est jugée acceptable. Le serveur met alors fin à la connexion avec un message de type `Alert` ;

1. L'authentification du client est possible, mais elle se fait de manière indépendante.

— dans le cas contraire, le serveur choisit une suite cryptographique parmi celles proposées par le client et émet le message `ServerHello` qui fait état de son choix. Ce message contient également une valeur aléatoire, nommée `server_random`.

Le serveur envoie ensuite son certificat (dans le message `Certificate`) et envoie un message `ServerHelloDone` pour indiquer qu'il attend maintenant une réponse du client.

Dans l'exemple donné, on suppose que le serveur choisit la suite `TLS_RSA_WITH_AES_128_CBC_SHA` :

— `RSA` décrit ici à la fois la méthode d'authentification du serveur et la méthode d'établissement des secrets de session : une fois que le client aura reçu et vérifié le certificat du serveur, il tirera un secret de session au hasard, le *pre-master secret*, et le chiffrera en utilisant la clé publique contenue dans le certificat. Cet aléa chiffré sera ensuite envoyé dans le message `ClientKeyExchange`, que seul le serveur pourra déchiffrer. Il s'agit donc d'une authentification implicite du serveur ;

— `AES_128_CBC` indique que l'algorithme de chiffrement par bloc AES va être utilisé en mode CBC avec une clé de 128 bits pour chiffrer le canal de communication ;

— `SHA` concerne enfin la protection en intégrité du canal de communication : HMAC SHA-1 sera employé.

Fin de la négociation Une fois la suite choisie et le certificat reçu, le client vérifie la chaîne de certification. Si le certificat n'est pas validé, le client émet une alerte qui met fin à la connexion. Sinon, il poursuit et envoie un message, `ClientKeyExchange`, contenant le *pre-master secret* chiffré.

À partir de là, le client et le serveur disposent tous les deux du *pre-master secret* (le client l'a tiré au hasard, et le serveur peut déchiffrer le message `ClientKeyExchange`), ainsi que d'éléments aléatoires publics échangés lors des messages `ClientHello` et `ServerHello`. Ces éléments partagés sont alors dérivés pour fournir les clés symétriques qui seront utilisées pour protéger le trafic en confidentialité et en intégrité.

Des messages `ChangeCipherSpec` sont échangés pour indiquer l'activation des paramètres (algorithmes et clés) négociés. Les messages `Finished` sont donc les premiers à être protégés cryptographiquement, et contiennent un haché de l'ensemble des messages échangés pendant la négociation, afin de garantir a posteriori l'intégrité de la négociation.

2 Attaques protocolaires liés à la phase de négociation

Commençons par nous intéresser aux problèmes de sécurité liés à la phase de négociation.

2.1 En finir avec SSL

De nombreuses vulnérabilités sont connues sur SSLv2 : négociation à la baisse (*downgrade attack*), réutilisation des clés pour le chiffrement et la protection en intégrité, algorithmes cryptographiques faibles... Il est communément accepté que cette version du protocole ne doit plus être utilisée [56].

SSLv3, qui a bientôt 20 ans, a longtemps bénéficié d'un sursis. Cependant, les piles SSL/TLS ne supportant que SSLv3 sont aujourd'hui des antiquités ne proposant pas tout le confort moderne apporté par les versions de TLS plus récentes. En effet, de telles implémentations ne supportent pas les extensions (et y sont parfois intolérantes). Étant donné le rôle de plus en plus important des extensions dans la sécurité de SSL/TLS, cette absence est aujourd'hui inacceptable.

Avec POODLE [40], décrit dans la section 3, Bodo Möller et Adam Langley ont donné une nouvelle raison d'abandonner SSLv3. Des changements sont en cours, au sein de l'IETF [4] et dans différentes implémentations, pour retirer définitivement le support de SSLv3. Du point de vue de la compatibilité, il existe encore quelques rares serveurs HTTPS accessibles uniquement en SSLv3 ; de même, Internet Explorer 6, dans sa configuration par défaut, ne parle pas TLS 1.0. Il est sans doute largement temps de mettre à jour ces systèmes.

À partir de maintenant, on peut donc cesser de parler de SSL/TLS, pour ne parler que de TLS ! Des voix s'élèvent même à l'IETF pour marquer également TLS 1.0 (voire TLS 1.1) comme obsolète.

SSLv2 **et SSLv3** doivent être abandonnés.

Il existe essentiellement deux grandes familles d'échange de clé :

— le chiffrement RSA, qui repose sur PKCS#1 v1.5, vulnérable à des attaques décrites par Bleichenbacher en 1998 [12]. De plus, cette méthode ne garantit pas la *forward secrecy* (FS [2]) : l'obtention de la clé privée à un instant donné permet de déchiffrer les communications passées ;

2. On parle aussi parfois de *Perfect Forward Secrecy* ou PFS.

— un échange de clé Diffie-Hellman éphémère (sur corps fini ou sur courbes elliptiques), signé côté serveur. Les suites cryptographiques contenant `DHE` ou `ECDHE` assurent la FS.

Les suites offrant la *forward secrecy* doivent être favorisées.

2.2 Renégociation et reprise de session

En 2009, Ray et Dispensa ont décrit une attaque sur la renégociation [45]. TLS dispose en effet d'un mécanisme permettant, au milieu d'une connexion TLS, de renégocier l'ensemble des paramètres de sécurité. Les seuls usages légitimes de la renégociation sont :

— rafraîchir les clés ;
— demander une authentification client après l'établissement de la connexion (par exemple suite à une requête HTTPS demandant accès à une ressource privilégiée).

En dehors de ces usages, la renégociation peut en théorie permettre de renégocier la version, la suite cryptographique, ou tout autre paramètre véhiculé par des extensions. Une telle flexibilité n'a aucun sens en pratique, et est au contraire à l'origine de plusieurs faiblesses.

La spécification initiale n'offrait aucune garantie quant au lien entre les messages échangés avant et après la renégociation : dans certains cas, il était possible de créer une confusion, permettant certaines attaques [60]. Pour cette raison, une extension a été spécifiée [48], afin d'ajouter ce lien entre une négociation et la précédente.

En 2014, l'équipe INRIA Prosecco a mis au jour une nouvelle attaque, *Triple Handshake* [8], similaire à l'attaque de 2009. Elle repose non seulement sur l'utilisation de la renégociation, mais également sur la reprise de session. Si un serveur légitime S met à la fois en œuvre la renégociation et la reprise de session, un attaquant contrôlant un serveur avec un certificat reconnu par le client C peut injecter un préfixe clair choisi au sein d'une connexion entre C et S.

La figure 2 décrit la première étape de l'attaque. On suppose que la victime C se connecte à l'attaquant A, qui dispose d'un certificat légitime pour un site donné. En parallèle de la négociation TLS entre C et A, ce dernier entame une connexion vers un serveur légitime S. A réutilise alors le `client_random` de C auprès de S, puis réutilise le `serveur_random` de S auprès de C. On suppose de plus que le mécanisme d'échange de clé utilisé pour les deux connexions est le chiffrement RSA [3]. Comme A est

3. L'article présente des attaques similaires dans le cas où un échange Diffie-Hellman est utilisé, qui fonctionnent sous certaines conditions.

le serveur légitime auquel C se connecte, il peut déchiffrer le *pre-master secret* et le réutiliser auprès de S. Le résultat net de cette manipulation est que les deux communications (à gauche et à droite du schéma), partagent un même *master secret* ! En effet, dans la spécification initiale, la dérivation se fait uniquement à partir du *pre-master secret* et des aléas transmis par le client et le serveur.

Une fois que deux sessions TLS partagent un même *master secret*, plusieurs attaques sont envisageables. L'une d'elle consiste pour l'attaquant à reprendre la session à gauche et à droite, d'injecter des messages clairs auprès de S, puis de forcer une renégociation. C'est le schéma décrit à la figure 3. Ainsi, C commence une connexion avec A, puis reprend une autre session avec S. La renégociation sécurisée ne prenant pas en compte la reprise de session, le cas n'est *pas* couvert. De plus, avant la publication de l'attaque, la plupart des navigateurs acceptaient sans sourciller un changement d'identité (les certificats présentés par A et S sont différents) au cours d'une session TLS. Comme pour l'attaque sur la renégociation, l'impact sur la sécurité dépend beaucoup du protocole véhiculé par TLS, mais certains scénarios d'attaque réalistes ont été proposés pour HTTPS.

Pour contrer cette attaque, les chercheurs ont proposé de réviser la manière de dériver le *master secret*, et donc les clés utilisées pour protéger le trafic (voir figure 4). La spécification en cours de standardisation [7] ajoute ainsi l'ensemble des messages passés échangés pendant la négociation à l'étape de dérivation, garantissant que des sessions réalisées avec des serveurs différents résulteront en des secrets distincts.

Il existe un autre atout important de cette nouvelle méthode de dérivation des clés : les paramètres proposés et négociés sont désormais liés cryptographiquement au *master secret*, rendant caduques plusieurs autres attaques, comme les *cross-protocol attacks*, jouant sur la confusion entre différents messages [58,37], ou plus récemment l'attaque FREAK (voir section 5.8).

La correction décrite dans la RFC 5746 doit être mise en œuvre sur les clients et les serveurs.

Lorsque l'extension `session-hash` sera définie, elle devra être mise en œuvre.

En attendant, des restrictions doivent être mises en place pour éviter que la vulnérabilité soit exploitable : restrictions sur la renégociation ou la reprise de session, vérification des identités plus strictes par les clients.

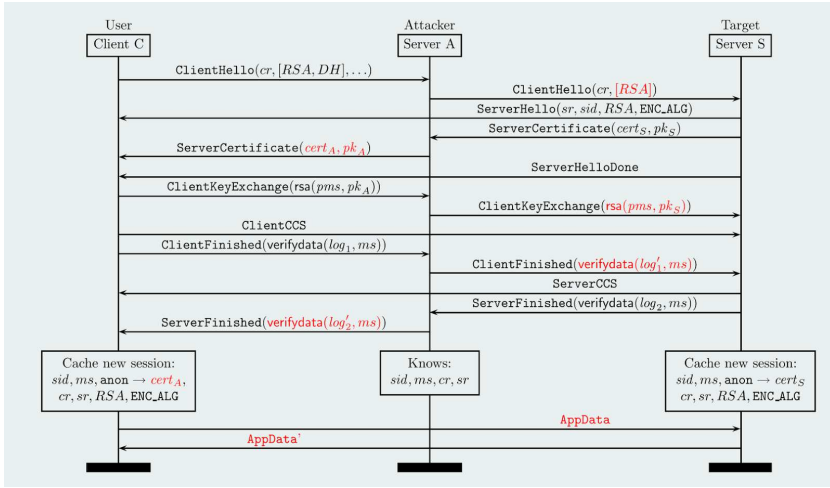


Figure 2. Première étape de l'attaque *Triple Handshake* : l'objectif est d'obtenir deux sessions TLS partageant un même *master secret*. Source : équipe INRIA Prosecco.

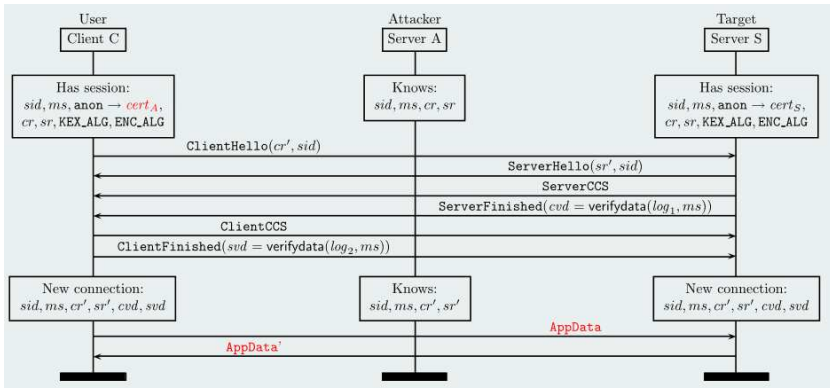


Figure 3. Seconde étape de l'attaque *Triple Handshake* : injection de messages clairs choisis au cours d'une connexion entre C et S. Source : équipe INRIA Prosecco.

Schéma de dérivation historique dans TLS :

```
MS = PRF (PMS, "master secret",
          ClientHello.random + ServerHello.random)
```

Schéma proposé pour contrer le *Triple Handshake* :

```
MS = PRF (PMS, "master secret",
          H(Handshake messages))
```

Figure 4. Schémas de dérivation du *master secret* (MS) à partir du *pre-master secret*, issu de l'algorithme d'échange de clé. La dérivation utilise une *Pseudo-Random Function* (PRF), construite à partir de fonctions de hachage.

2.3 Mécanisme de *fallback*

Afin de pouvoir communiquer avec des implémentations de SSL/TLS non standard, il est courant, dans les navigateurs, lors de l'échec d'une connexion SSL/TLS, de retenter une connexion avec des paramètres différents. Jusqu'à récemment, on pouvait voir des tentatives utilisant TLS 1.2, puis TLS 1.1, puis TLS 1.0 et enfin SSLv3 sans extension.

Le problème principal de ces connexions répétées, c'est qu'un attaquant peut aisément les déclencher, puisque l'échec de connexion n'est *pas* authentifié : il lui suffit d'envoyer une alerte TLS en clair ou un `RST` au niveau TCP. L'ennui, c'est que la nouvelle connexion ne fait plus état des réelles capacités du client : l'attaquant provoque ainsi en pratique une négociation à la baisse.

Il existe normalement des indicateurs pour détecter ce genre de négociation à la baisse, mais ils ne fonctionnent que pour l'échange de clé RSA [4], et sont parfois mal implémentés [5].

Cette technique a été présentée lors de la publication de l'attaque POODLE, puisqu'elle permet à un client et un serveur TLS récents de négocier SSLv3 en présence d'un attaquant réseau.

Une proposition pour contrer les *fallbacks* abusifs vient d'être standardisée [38] :

- lorsqu'un client tente une nouvelle connexion auprès du serveur avec une version inférieure, il ajoute un indicateur (une suite cryptographique factice `FALLBACK_SCSV`) ;
- si le serveur reçoit un `ClientHello` proposant une version inférieure à la version maximale qu'il supporte, et contenant l'indicateur `FALLBACK_SCSV`, il doit mettre fin à la connexion.

Dans la mesure du possible, les mécanismes de *fallback* doivent être abandonnés.

Si ces mécanismes sont nécessaires, il faut implémenter le mécanisme `FALLBACK_SCSV` pour contrer les négociations à la baisse.

4. Lors d'un échange de clé par chiffrement RSA, le client transmet le *pre-master secret* chiffré au serveur. Ce secret est constitué de 48 octets, dont les deux premiers doivent contenir la version *maximale* supportée par le client.

5. Par exemple, certaines versions de Windows vérifient que la version intégrée au *pre-master secret* correspond à la version négociée, alors que l'objectif est de pouvoir annoncer une version plus grande.

3 Nouvelles vulnérabilités sur le *Record Protcol*

Cette section décrit cinq attaques publiées entre 2011 et 2014 sur la protection des flux applicatifs. Chacune d'entre elles a fait l'objet d'une preuve de concept visant à récupérer un cookie de session HTTPS. La figure 5 présente les différentes manières dont les données applicatives peuvent être protégées : avec un *streamcipher*, avec un *blockcipher* (tous deux couplés avec un MAC) ou avec un chiffrement authentifié.

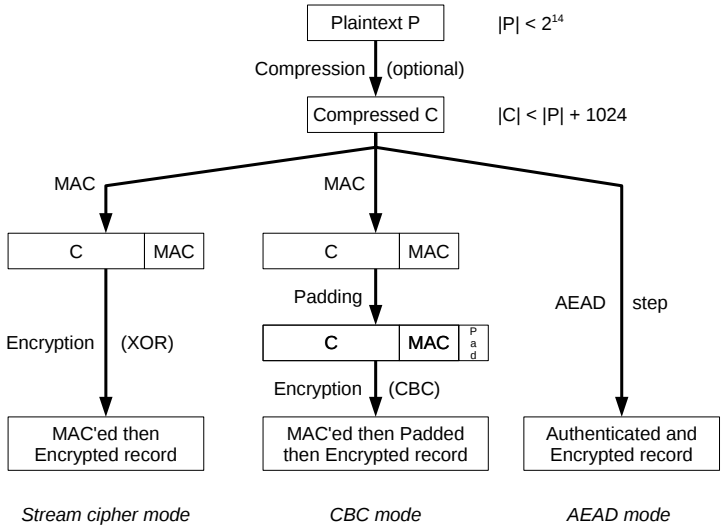


Figure 5. Les différents modes du *Record Protocol*.

3.1 BEAST

En 1995, Rogaway a décrit une attaque à clair choisi adaptatif contre le mode CBC, dès que l'IV utilisé est prédictible [50]. En 2002, Möller a remarqué que TLS 1.0 remplissait les conditions [39]. Cependant, l'attaque n'était pas considérée réaliste, étant donné les hypothèses nécessaires (on suppose tout de même que l'attaquant maîtrise partiellement le clair). La situation a changé en 2011 avec BEAST (*Browser Exploit Against SSL/TLS*), présenté par Duong et Rizzo [17]. La figure 6 (à gauche) détaille l'étape de chiffrement réalisé lorsque TLS 1.0 utilise le mode CBC.

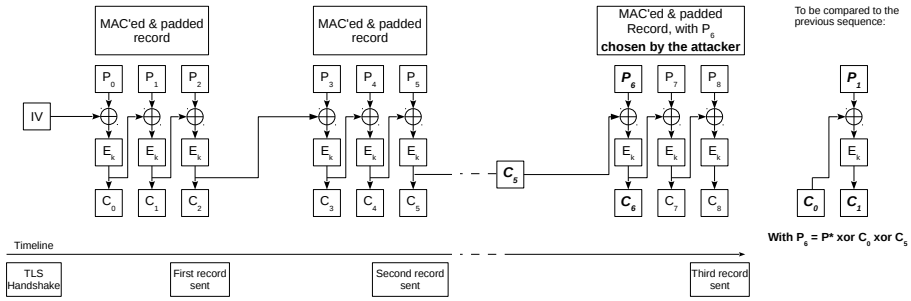


Figure 6. [À gauche] Utilisation du mode CBC avec IV implicite dans TLS 1.0 : le premier IV est généré lors de la phase de négociation, puis tous les messages sont chiffrés comme un flux CBC continu. [À droite] Pour tester si $P_1 = P^\star$, l'attaquant envoie un message clair commençant par P_6, puis compare le résultat C_6 au bloc C_1, observé précédemment.

En utilisant les notations de la figure 6, l'attaquant peut *tester* si la valeur de P_1 est égale à une valeur $P^\star$ de son choix. En effet, lorsque les deux premiers *records* ont été envoyés, il peut observer C_5, et savoir que ce sera le prochain IV. Pour tester son hypothèse, il lui suffit d'envoyer le bloc clair $P_6 = P^\star \oplus C_5 \oplus C_0$ et d'observer $C_6 = E(P_6 \oplus C_5) = E(P^\star \oplus C_0)$. Si le choix était bon, (c'est-à-dire si P_0 vaut $P^\star$), alors C_6 sera égal à C_1. Une telle tentative est décrite sur la droite de la figure.

De plus, pour éviter de devoir deviner un bloc en entier, la preuve de concept présentée utilisait une astuce : aligner le bloc P_1 à deviner de manière à ce qu'un seul octet soit inconnu. Par exemple, si l'attaquant sait que P_1 vaut `":SESSION_TOKEN=?"`, où `?` est l'octet inconnu, il lui suffit de 128 essais en moyenne pour retrouver l'octet (voire moins, si l'octet à retrouver appartient à un *charset* contraint).

En pratique, pour envoyer le bloc P_6 au moment de son choix, l'attaquant doit pouvoir injecter de manière précise du clair dans le même tunnel que l'application légitime, ce qui nécessite de contourner la *Same Origin Policy* (SOP [6]) [5].

La meilleure contre-mesure à cette attaque est de rendre l'IV explicite pour chaque *record*, comme le spécifie TLS 1.1. On peut aussi utiliser un *streamcipher* pour éviter d'utiliser le mode CBC, mais cela revient en pratique à utiliser RC4, ce qui pose d'autres problèmes (voir section 3.4). Un bricolage consistant à découper les *records* de longueur n en deux

6. La *Same Origin Policy* est un mécanisme de sécurité fondamental implémenté dans les navigateurs, dont l'objectif est d'empêcher un script chargé depuis un site donné d'interagir librement avec un un autre site.

records de longueur 1 et $n-1$ a été implémenté dans les navigateurs pour rendre l'attaque inefficace, même avec TLS 1.0.

3.2 CRIME, TIME et BREACH

En 2012, Duong and Rizzo ont publié une autre attaque intitulée CRIME (*Compression Ratio Info-leak Made Easy*) [49]. Cette fois encore, l'objectif était de récupérer la valeur d'un cookie d'authentification. L'attaque repose sur l'étape de compression, et suppose que l'attaquant peut choisir une partie du texte clair envoyé en même temps que le cookie, par exemple le chemin dans l'URL. L'année suivante, une autre équipe de recherche a présenté TIME (*Timing Info-leak Made Easy*) attack [55], une variante de CRIME, utilisant une méthode d'observation différente.

Supposons que `SESSION_ID`, le cookie de session, est une chaîne hexadécimale, et que l'attaquant peut déclencher des requêtes HTTPS successives en contrôlant une partie de l'URL, par exemple des requêtes de la forme `www.target.com/SESSION_ID=X`, avec `X` un caractère hexadécimal. Comme le contrôle sur le clair est très lâche, cette hypothèse ne nécessite pas de violer la *Same Origin Policy*. Lorsque la requête est compressée, la redondance est maximale lorsque l'attaquant a choisi le bon caractère hexadécimal, ce qui va résulter en une compression plus efficace. Dans certains cas, le *record* correspondant va être plus petit d'un octet, ce qui est observable [7].

CRIME et TIME utilisent deux méthodes différentes pour observer la différence de taille du *record*. CRIME suppose que l'attaquant peut observer la taille des paquets chiffrés sur le réseau. TIME en revanche mesure le délai de transmission des réponses. En effet, en alignant la taille du *record* avec la taille de la fenêtre TCP, il est possible qu'une variation de taille, même d'un octet, force le client à attendre un ACK TCP avant d'envoyer l'octet restant, ce qui est facilement mesurable depuis le navigateur (un aller-retour avec le serveur, ou *Round Time Trip* prend de l'ordre de 100 ms).

De manière similaire, les auteurs de l'attaque TIME ont proposé une attaque pour récupérer des éléments secrets envoyés de manière répétée par le serveur, comme les jetons anti-CSRF. En 2013, une adaptation de cette dernière attaque, utilisant la compression HTTP pour récupérer les jetons émis par le serveur, a été présentée : BREACH [43].

7. Pour cela, certains paramètres doivent être ajustés, comme l'alignement (en cas d'utilisation d'un *blockcipher*) ou encore une méthode fiable pour réinitialiser le dictionnaire de compression qui a un état.

La seule contre-mesure efficace pour les attaques utilisant la compression TLS est de désactiver cette fonctionnalité. Pour l'attaque côté serveur reposant sur la compression HTTP, il est nécessaire de restreindre la compression HTTP lors des requêtes émises par un site tiers ; pour cela, il faut vérifier l'en-tête *Referer*. En effet, il est impossible en pratique de désactiver complètement la compression HTTP, sous peine d'augmenter la bande passante nécessaire à de nombreuses communications.

3.3 Lucky 13

Lorsque TLS est utilisé avec un *blockcipher*, un motif d'intégrité est calculé sur le clair à l'aide d'un HMAC, puis accolé au message clair ; le résultat est ensuite aligné à la taille d'un bloc (*padding* ou bourrage), et enfin, la primitive de chiffrement par bloc est appelée en mode CBC. Ce paradigme, *MAC-then-Encrypt* est connu pour permettre des attaques : les *Padding Oracle attacks*, décrites par Vaudenay en 2002 [57]. Dès qu'un attaquant peut distinguer une erreur liée au *padding* d'une erreur liée au motif d'intégrité, que ce soit à l'aide d'un message d'erreur accessible à l'attaquant, ou à cause d'une différence dans le temps de traitement, de l'information fuit, ce qui peut permettre de récupérer le texte clair.

Lors du déchiffrement des *records*, le récepteur doit vérifier que le *padding* est correct : les p derniers octets du bloc doivent avoir la même valeur $p-1$. Ainsi, les blocs terminant par 00, 01 01 ou encore 02 02 02 sont acceptables. Ensuite, le motif d'intégrité est extrait et vérifié.

On note $P = p_0 p_1 \cdots p_{n-1}$ un message clair après *padding*, et $C = c_0 c_1 \cdots c_{n-1}$ le chiffré correspondant (voir Fig. 7, à gauche). Pour deviner la valeur de p_{n-1}, un attaquant peut forger un chiffré contenant deux blocs, $C^\star C$, avec C le bloc chiffré à décrypter, et $C^\star$ un bloc aléatoire dont le dernier octet est $c^\star_{n-1} = g$ (le déchiffrement de ce second bloc est décrit à droite sur la figure 7). Si g est égal à $p_{n-1} \oplus IV_{n-1}$, alors le résultat du déchiffrement $E_k^{-1}(C^\star C)$ se terminera par un octet nul, et le message sera considéré valide, ce qui mènera à une erreur sur le motif d'intégrité. Sinon, le *padding* sera incorrect avec forte probabilité [8].

Si l'attaquant peut distinguer entre une erreur d'intégrité et une erreur de *padding*, ce dernier cas permet de détecter le contenu du bloc, octet par octet. En effet, une fois le dernier octet p_{n-1} identifié, il suffit de recommencer avec un bloc $C^\star$ terminant par $(g)|(p_{n-1} \oplus 01)$ pour déterminer $p_{n-2} = g \oplus 01 \oplus IV_{n-2}$, et ainsi de suite.

8. Pour être précis, il y a une probabilité de 2^{-16} que le déchiffré obtenu se termine par 01 01, de 2^{-24} pour 02 02 02, etc. Pour éliminer ces faux positifs, il suffit de répéter l'opération avec un $C^\star$ différent.

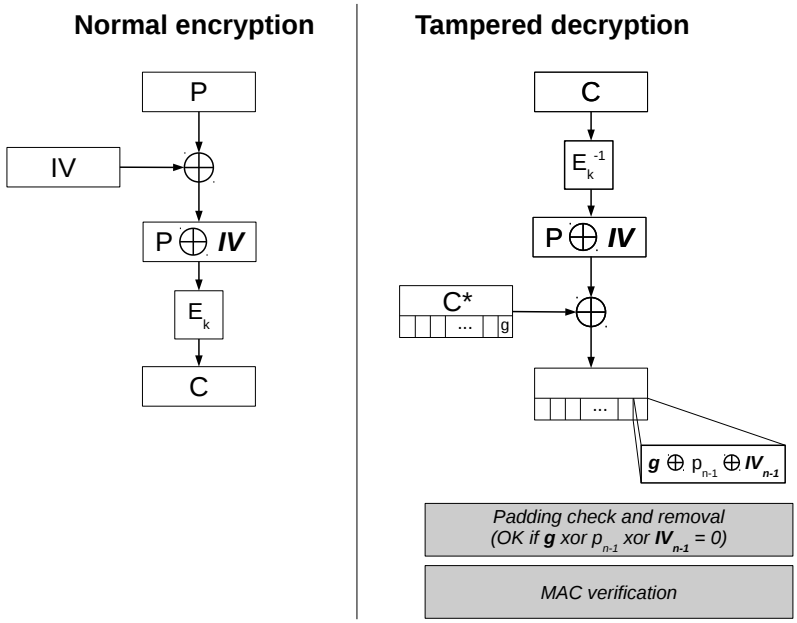


Figure 7. Chiffrement CBC, et déchiffrement dans le cadre d'une attaque par oracle de *padding*. C, $C^\star$ (en particulier son dernier octet g) et IV sont connus de l'attaquant.

Concrètement, l'attaquant doit exploiter des différences dans le temps de traitement. En effet, si le *padding* est invalide, le MAC n'est pas vérifié et le serveur répond plus rapidement. C'est pourquoi TLS 1.1 [15] indique que les implémentations doivent s'assurer que le temps de traitement des *records* doit être essentiellement le même dans tous les cas [9].

De plus, ces attaques mettent fin à la connexion TLS, ce qui les rend a priori peu intéressantes. En 2012, des chercheurs de l'université de Londres (Royal Holloway) ont étudié l'applicabilité de l'attaque à DTLS [41]. *Datagram TLS* (DTLS) [47] est similaire à TLS, mais repose sur UDP et non sur TCP ; comme UDP ne garantit pas un transport des paquets fiable, les *records* peuvent être corrompus ou perdus. C'est pourquoi DTLS ne met pas fin à la connexion en cas d'erreur de déchiffrement ou d'intégrité. Les auteurs ont montré qu'une *timing attack* était possible sur DTLS pour obtenir un oracle distinguant les erreurs d'intégrité des erreurs de *padding*.

En réalité, les mêmes chercheurs ont montré en 2013 qu'une attaque similaire était possible sur TLS en pratique [2], mais qu'il était nécessaire

9. *Implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct.*

que le secret à retrouver soit répété dans différentes sessions TLS (chaque essai cassant la connexion en cours). La preuve de concept a été nommée Lucky 13, où 13 est la taille de l'en-tête intégré au calcul du motif d'intégrité.

3.4 Biais statistiques de RC4

RC4 est un *streamcipher* conçu par Rivest en 1987. Cette primitive est très simple à implémenter et a de très bonnes performances. C'est pourquoi elle a été intégrée à de nombreux protocoles (dont le WEP pour la protection WiFi, et TLS). Depuis 1995, plusieurs biais statistiques ont été identifiés sur les premiers octets de la suite chiffrante produite par RC4. Ces biais ont notamment mené à des attaques très efficaces sur WEP [54].

En 2013, deux équipes de recherche ont présenté des attaques pratiques permettant de retrouver un texte clair s'il avait été chiffré un grand nombre de fois avec différentes clés [31,1] (cas classique d'un cookie de session HTTPS). Une des attaques présentées repose sur des biais très forts des 256 premiers octets de la suite chiffrante. Les chercheurs ont en effet produit 2^{45} suites chiffrantes correspondant à des clés différentes pour obtenir des statistiques de référence. Ensuite, en obtenant un grand nombre de chiffrés d'un même clair avec des clés différentes, il est possible de retrouver le clair en minimisant la distance entre le profil statistique de référence et le profil de la cible. En fonction de la position dans la suite chiffrante, ce grand nombre varie de 2^{24} à 2^{32} messages.

La figure 8 montre un exemple de distribution sur le 50^{e} octet de clair dont on sait qu'il vaut `"A"`, `"X"` ou `"."`[10]. Le premier schéma décrit le profil de référence (Z50). Après avoir récolté 2^{30} chiffrés de `A` (`0x41`) à la position 50, les petits graphes montrent quelle aurait été la distribution de la suite chiffrante, en faisant l'hypothèse que le clair était `"A"` (`0x41`), `"X"` (`0x58`) ou `"."` (`0x2e`). Il apparaît clairement que le premier est le plus proche de la distribution de référence (les pics sont alignés), ce qui fait de `A` le meilleur candidat.

L'attaque est compliquée à implémenter, puisqu'elle requiert de nombreuses connexions TLS, et que seules les données placées au début du flux peuvent être retrouvées. Pour aller plus loin, les chercheurs ont utilisés d'autres biais sur des octets consécutifs de la suite chiffrante, décrits par Fluhrer et McGrew [21]. L'attaque résultant de l'exploitation de ces biais requiert plus de données échangées, mais pas nécessairement au début de la connexion HTTPS, rendant une preuve de concept réaliste.

10. Bien entendu, l'attaque ne nécessite pas de telles conditions pour réussir. Il s'agit simplement d'un exemple visuel.

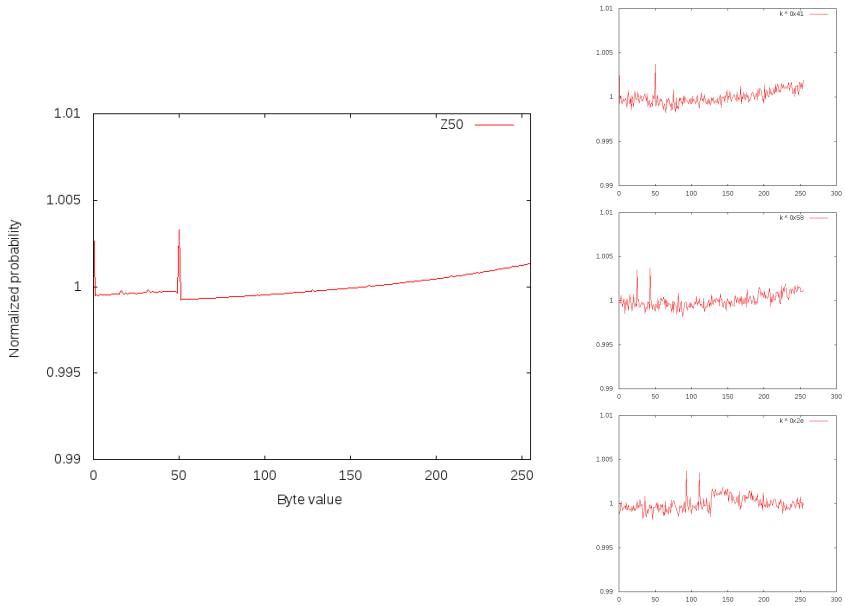


Figure 8. [À gauche] Distribution statistique de Z50, le 50^{e} octet de la suite chiffrante. Source : [1]. [À droite] Reconstruction statistique de Z50 à partir de 2^{30} messages, en supposant que le caractère clair est `A`, `X` ou `.`).

Depuis, de nouveaux travaux ont permis d'améliorer l'attaque [23] : la récupération de mots de passe peut désormais se faire avec 2^{26} connexions. En parallèle, Mantin a montré début 2015 qu'une attaque théorique qu'il avait présenté 13 ans plus tôt était exploitable sur TLS [32] : le résultat permet, à partir de 2^{24} messages, d'obtenir de l'information sur certains bits de la suite chiffrante. Enfin, l'IETF a publié début 2015 une RFC interdisant l'utilisation de RC4 dans TLS [42], pour envoyer un signal fort quant aux dangers de cet algorithme.

3.5 POODLE

En octobre 2014, Möller, Duong et Kotowicz ont présenté POODLE (*Padding Oracle on Downgraded Legacy Encryption*) [40], une nouvelle attaque sur le *padding* CBC utilisé dans SSLv3. En effet, SSLv3 utilise une méthode de *padding* différente de TLS : quand n octets doivent être ajoutés pour compléter un bloc, *seul le dernier octet du bloc* doit contenir $n-1$, mais les autres octets de *padding* peuvent prendre des valeurs arbitraires. Un attaquant peut exploiter ce laxisme dans la vérification pour obtenir un oracle de *padding*.

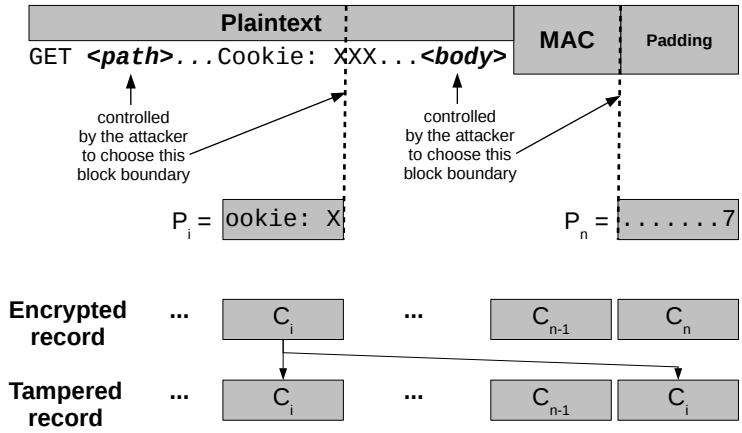


Figure 9. Exemple d'une attaque POODLE, avec une taille de bloc de 8 octets.

Supposons que l'attaquant peut déclencher des requêtes successives à un site, en mode CBC avec SSLv3. En particulier, comme pour CRIME, on suppose que l'attaquant contrôle une partie de l'URL. Il lui est donc possible d'aligner la fin du cookie d'authentification sur une fin de bloc. De plus, on suppose que l'attaquant peut intégrer un corps arbitraire à la requête émise (qui se situera après les en-têtes HTTP) ; il est donc capable d'aligner le message clair (en incluant le MAC) avec une autre fin de bloc (voir la figure 9). Ainsi, l'attaquant *sait* qu'un bloc entier va devoir être ajouté lors du *padding*. Or, avec SSLv3, la seule contrainte sur ce bloc clair est qu'il doit finit par un octet à $n-1$, où n est la taille du bloc.

Une fois la requête ainsi forgée transmise par le navigateur, l'attaquant intercepte le message émis et le modifie. Il remplace le bloc final (qui ne contient que du *padding* par le bloc dont il souhaite deviner le dernier octet, comme le montre la figure. Si le déchiffrement par le serveur mène à une valeur correcte (si le dernier octet vaut $n-1$), le reste du bloc est ignoré et le *record* est accepté par le serveur. L'attaquant peut donc en déduire que $E_k^{-1}(C_i) \oplus C_{n-1}$ vaut $n-1$, et donc que le dernier octet du bloc clair P_i vaut $C_{n-1} \oplus C_{i-1} \oplus (n-1)$. Si le *padding* ne se termine pas par $n-1$, le chiffrement échouera lors de la vérification du MAC, et mettra fin à la session. L'attaquant peut alors réessayer, avec une nouvelle chance de succès puisque le bloc chiffré C_{n-1} randomise le déchiffrement du dernier bloc. Ainsi, chaque essai a une probabilité de 2^{-8} : chaque octet du cookie peut être retrouvé avec en moyenne 256 requêtes.

En théorie, l'attaque ne fonctionne qu'avec SSLv3, ce qui implique dans le cas général que l'attaquant doit commencer par forcer le client à négocier cette vieille version du protocole (les mécanismes de *fallback* ont été décrits dans la section 2.3). En pratique, certaines implémentations TLS 1.0 ne suivent pas la norme et réutilisent le *padding* SSLv3, les rendant vulnérables à la même attaque [11].

3.6 Discussion sur le *Record Protocol*

Concernant le mode CBC, il est intéressant de se rendre compte que toutes les attaques (BEAST, Lucky 13 et POODLE) avaient été décrites en 2001 par Bodo Möller dans un fichier texte disponible sur le site d'OpenSSL [12]). Il a cependant fallu que les preuves de concept soient publiées pour que des mesures soient prises.

Il ressort de toutes ces attaques que RC4 doit être évité, ainsi que le mode CBC. L'ennui, c'est que seul TLS 1.2 offre une alternative à ces deux modes de protection du *Record Protocol*, avec le mode combiné GCM (ou CCM). Il est donc essentiel d'accélérer la transition vers TLS 1.2. En effet, une autre solution est d'utiliser des rustines complexes (le lecteur intéressé pourra par exemple lire le billet de blog d'Adam Langley décrivant le déchiffrement en temps constant du mode CBC dans OpenSSL [13]. Récemment, une extension TLS pour activer le paradigme *Encrypt-then-MAC* (qui contre efficacement les attaques contre le *padding* CBC) a été publiée [26], mais elle promet d'être aussi difficile à déployer que TLS 1.2.

Enfin, un mécanisme de défense en profondeur concernant HTTPS serait d'éviter la répétition des secrets transmis au sein de sessions TLS. C'est l'objet de travaux de recherche présentés lors de la conférence ASIACCS [36].

TLS 1.2 doit être déployé et privilégié dans les négociations.

La compression, RC4 et SSLv3 ne doivent plus être utilisés.

Si TLS 1.2 ne peut être négocié, des mécanismes spécifiques doivent être mis en œuvre pour prendre en compte les attaques BEAST et Lucky 13 : éclater les *records* $(1/n-1)$, déchiffrement CBC en temps constant.

11. `https://www.imperialviolet.org/2014/12/08/poodleagain.html`
12. `https://www.openssl.org/~bodo/tls-cbc.txt`
13. `https://www.imperialviolet.org/2013/02/04/luckythirteen.html`

4 Évolution du modèle de confiance : menaces et solutions

4.1 Problème de validation de la chaîne de certificats

L'histoire des erreurs d'implémentations lors de la vérification de la chaîne de certificats est longue... et a fait l'objet de nouvelles vulnérabilités en 2014 (voir par exemple la section 5.1).

Un autre problème majeur lié à la vérification des certificats est l'omission pure et simple de l'appel aux fonctions correspondantes. Des travaux [20] ont montré qu'une part importante des applications sur *smartphones* ne valident *pas* la chaîne de certificat présentée par le serveur, annulant toute garantie quant à l'identité du serveur : un attaquant réseau actif peut alors répondre à la place du serveur légitime. Le problème est parfois plus subtil, puisque c'est l'API qui est mal utilisée [24] : avec `libcurl`, mettre `CURLOPT_SSL_VERIFYHOST` à 2 provoque l'échec des connexions non authentifiées¸ alors que la valeur 1 (ou `TRUE` dans certains cas) correspondait à une option de *debug* acceptant en pratique toutes les connexions... D'autres API souffrent de problèmes similaires, comme en témoigne l'exemple de GnuTLS de la section 5.2.

Lors du développement d'une application mettant en œuvre TLS, il faut s'assurer que la validation des certificats est effective, en consultant la documentation des bibliothèques utilisées, mais surtout en testant que des connexions devant échouer échouent.

4.2 Cryptographie faible

Les chaînes de certification mettent en œuvre de la cryptographie asymétrique. Pour garantir un niveau de sécurité acceptable, il faut utiliser des primitives et des tailles de clés conformes à l'état de l'art [3]. En particulier, les clés RSA doivent avoir une taille supérieure à 2048 bits. Ainsi, plusieurs éditeurs logiciels (dont Google et Mozilla) prévoient de rejeter à terme les certificats avec des clés RSA 1024 bits.

Du côté des fonctions de hachage, on savait MD5 faible depuis 2005 [59,34], avec une application concrète en 2009 ayant permis la signature frauduleuse d'une autorité de certification intermédiaire [53]. Depuis 2005, SHA-1 est aussi considérée comme affaiblie, et des résultats récents

montrent que les premières collisions SHA-1 ne devraient plus tarder [14]. Une première décision, à l'initiative de Microsoft, a été prise par le CA/B Forum pour bannir SHA-1 pour les certificats expirant après fin 2016 [15]. Parmi les mesures concrètes mises en place, Google a annoncé une série de mesures [16] consistant à afficher des avertissements de plus en plus visibles pour les certificats expirant au-delà du 31 décembre 2015 (ce qui correspond à une politique plus sévère que la décision prise par le CA/B Forum). Les politiques de Microsoft [17] et de Mozilla [18] sont également disponibles en ligne.

Une clé RSA devrait aujourd'hui faire au moins 2048 bits.
Les algorithmes de hachage MD5 **et SHA-1** doivent être proscrits pour toute nouvelle signature.

Dans le cas de SHA-1, il faut dès à présent renouveler tous les certificats expirant au-delà du 31 décembre 2015.

4.3 Détection et reprise sur incident lors d'une émission d'un certificat illégitime

Depuis 2011, plusieurs incidents liés à l'émission de certificats illégitimes ont été rendus publics. Certains résultent d'attaques vite détectées (Comodo en 2011), ou d'attaques ayant mené à une exploitation réelle permettant l'usurpation de certains sites web (Diginotar en 2011 également). D'autres sont liés à des erreurs humaines lors de l'utilisation d'autorités intermédiaires dépendant d'autorités reconnues par les navigateurs (Türktrust en 2012, IGC/A en 2013).

Bien que le problème soit identifié, les solutions permettant de détecter, mais surtout de corriger ce genre d'incidents tardent à venir. La réponse classique à ce problème est la révocation, qui peut revêtir trois formes dans le cas de TLS :

14. Ainsi, des chercheurs ont récemment montré qu'une collision sur la fonction de compression SHA-1 réduite à 76 tours était possible (`https://marc-stevens.nl/research/sha1freestart/`). Rappelons que la fonction complète réalise 80 tours.

15. `https://cabforum.org/2014/10/16/ballot-118-sha-1-sunset/`

16. `http://googleonlinesecurity.blogspot.fr/2014/09/gradually-sunsetting-sha-1.html`

17. `http://social.technet.microsoft.com/wiki/contents/articles/1760.windows-root-certificate-program-technical-requirements-version-2-0.aspx`

18. `https://blog.mozilla.org/security/2014/09/23/phasing-out-certificates-with-sha-1-based-signature-algorithms/`

— les listes de révocations (CRL, définie dans la norme X.509 [14]), qui nécessitent d'échanger de grands volumes d'information de manière asynchrone, mais au prix d'une durée de validité souvent très longue (il n'est pas rare de tomber sur des listes valides pendant un an) ;
— les répondeurs OCSP (*Online Certificate Status Protocol*, décrit dans la RFC 6960 [51]) permettant d'obtenir le statut d'un certificat auprès de son émetteur. Cette solution pose des problèmes d'efficacité, puisque la vérification nécessite une connexion réseau en parallèle, et de vie privée, puisque les autorités de certification obtiennent les noms des sites visités ;
— *OCSP Stapling* est une extension TLS consistant à épingler une réponse OCSP fraiche, obtenue par le serveur lui-même, lors du dialogue TLS [18]. On résout ainsi les problèmes d'efficacité et de vie privée, mais cette méthode est encore trop peu répandue.

Enfin, un défaut commun à toutes ces méthodes est qu'en général, la révocation est facultative : si les informations de révocation ne sont pas disponibles, le client va généralement silencieusement supposer que tout va bien. C'est pourquoi lors d'incidents avérés, la seule solution efficace est d'ajouter des certificats dans une liste noire codée en dur dans les clients.

Depuis 2011, des propositions ont été faites pour améliorer la situation.

Gestion des CRL via les mises à jour logicielles Une méthode déployée par Google dans ses navigateurs est l'utilisation de listes de révocation compilées en amont, puis transmises aux navigateurs par un système de mises à jour régulières. Ces *CRLSets*[19] peuvent également contenir des informations pour la révocation en urgence de certificats de serveurs ou d'autorités.

La démarche permet une détection et un blocage efficace des certificats illégitimes, mais uniquement sur le périmètre concerné par les listes collectées par Google. En effet, il est impossible d'être exhaustif dans ce domaine, puisque le graphe des autorités pouvant émettre des CRL est immense, y compris en ne considérant que les autorités de confiance dans les navigateurs. De plus, un *CRLSet* avec une large couverture pourrait atteindre une taille énorme. C'est pourquoi la taille des CRL incluses est limitée à 250 Ko. En cas de dépassement, la CRL en question est retirée de la mise à jour, et l'autorité concernée informée.

19. `https://dev.chromium.org/Home/chromium-security/crlsets`

Depuis mars 2015, les produits Mozilla embarquent également une telle liste, appelée *OneCRL* [20], mais elle est pour l'instant limitée à la révocation des autorités intermédiaires, pour garder une taille raisonnable.

Il apparaît donc que cette solution est très partielle, mais sur le périmètre couvert, elle permet un déploiement fiable, éventuellement en urgence, d'informations sur la révocation.

Généralisation de l'épinglage OCSP *OCSP Stapling* est un mécanisme de révocation qui semble pertinent et efficace. Cependant, tant qu'il s'agit d'un mécanisme optionnel, il ne peut être utilisé de manière robuste et efficace. Adam Langley a ainsi comparé le mode *fail-safe open* (en cas d'échec, on laisse ouvert) actuellement mis en œuvre pour la révocation à une ceinture de sécurité qui se détachait lors d'un impact.

Pour permettre à certains sites d'annoncer qu'ils supportent l'*OCSP Stapling*, et que l'absence de cette extension TLS devrait être traitée comme une erreur fatale, une extension X.509 a été proposée [27] [21] : si un client rencontre l'extension dans un certificat X.509 sans que la connexion n'ait fait l'objet d'un épinglage OCSP, il faut mettre fin à la connexion.

Ainsi, seuls les sites annonçant dans leur certificat leur intention de supporter la fonctionnalité sont traités de manière sécurisée (et bloquante en cas d'erreur). Ce mécanisme est utile à combiner avec HSTS et HPKP (voir plus loin).

DANE Une autre solution, standardisée en août 2012 par l'IETF est DANE (*DNS-Based Authentication of Named Entities*, [30]). Ce nouveau protocole repose sur DNS pour transmettre des informations sur les certificats attendus pour une connexion TLS donnée, via de nouveaux enregistrements intitulés TLSA.

DANE offre différents paramètres sur la manière de désigner un certificat TLS donné. L'enregistrement peut porter sur le certificat serveur ou sur le certificat d'une autorité de la chaîne. De même, l'élément transmis par DANE peut être le certificat complet (ou son haché) ou bien la clé publique embarquée (ou son haché).

Il existe deux positionnements possibles de DANE vis-à-vis des IGC classiques TLS : soit en complément des vérifications actuelles, soit en remplacement. Dans le premier cas, l'idée est de contraindre encore plus la

20. `https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/`

21. Un en-tête HTTP aurait également pu être utilisé, comme pour HSTS et HPKP (voir plus loin), mais ce n'est pas la solution qui semble désormais retenue.

vérification, au prix d'une requête DNS(SEC) supplémentaire. Le second cas permet de sortir du modèle de confiance classique TLS/HTTPS, ce qui permet en théorie le déploiement de sites HTTPS sans avoir besoin d'acheter un certificat.

Dans les deux cas, la question de l'intégrité de l'enregistrement TLSA est essentielle, et doit reposer sur DNSSEC, ce qui n'est pas sans poser de nombreuses interrogations : déploiement du standard incertain, cryptographie limitée en pratique à RSA 1024, question de la confiance dans les clés racines, accès à des résolveurs validant de confiance et sécurisation du dernier kilomètre. Des discussions actuelles, il semble que DANE ne soit pas une solution viable à court ou moyen terme pour le monde HTTPS [22], d'autant plus que certains acteurs majeurs tels que Google [23] ne comptent pas l'implémenter.

En pratique, DANE semble applicable au monde SMTP, pour permettre aux entités déployant DNSSEC d'améliorer la situation actuelle en imposant l'usage de TLS là où la connexion en clair est aujourd'hui une solution de repli universellement déployée.

Restrictions X.509 appliquées aux autorités Le standard X.509 permet de restreindre la portée d'une autorité : l'autorité émettrice d'un certificat d'autorité peut embarquer dans le certificat émis des extensions contraignant le champ d'application des certificats. Il est alors possible de détecter et bloquer les certificats sortant du champ de compétence d'une autorité.

Par exemple, il est possible de restreindre une autorité à un ensemble de noms de domaine. Ce mécanisme a par exemple été mis en œuvre en 2014 pour que l'IGC/A, autorité destinée aux sites de l'administration française, ne soit valide que pour les domaines en `.fr`, `.re`, et autres territoires d'outre-mer français.

Cependant, cette solution ne peut s'appliquer de manière générale à l'ensemble des autorités de confiance des navigateurs, puisque les acteurs commerciaux peuvent légitimement signer des certificats pour n'importe quel nom de domaine.

***Certificate Pinning*, HSTS et HPKP** Le mécanisme habituellement appelé *Certificate Pinning* consiste à fixer un des certificats attendus dans la chaîne de certification émise par un site donné. En « épinglant »

22. `http://sockpuppet.org/blog/2015/01/15/against-dnssec/`
23. `https://www.imperialviolet.org/2015/01/17/notdane.html`

le certificat du serveur, il est ainsi possible de garantir qu'un certificat illégitime sera refusé. De même, il est possible d'épingler une autorité (racine ou intermédiaire), pour limiter les émetteurs acceptables pour un site donné.

Plusieurs implémentations existent déjà :

— dès 2011, Google a intégré une forme de *Certificate Pinning* à ses navigateurs, ce qui a permis entre autre de détecter la compromission de Diginotar. Un tel épinglage ne passait cependant pas à l'échelle et ne concernait que les sites web les plus connus ;

— Firefox permet par ailleurs depuis longtemps à l'utilisateur d'épingler les certificats des serveurs non reconnus, en créant des « exceptions permanentes ». On est alors dans un modèle proche de SSH (*Trust On First Use*, TOFU) ;

— certaines applications mobiles liées à un service web distant identifié peuvent utiliser les fonctions de validation standard des systèmes, en épinglant le certificat attendu.

En 2012, l'en-tête HTTP *Strict Transport Security* (HSTS [29]) a été standardisé. Si un site web contacté en HTTPS présentait cet en-tête, il lui était possible d'indiquer au navigateur que toutes les communications futures le concernant devraient être faite en HTTPS (pendant une certaine durée), sans possibilité de contourner une erreur éventuelle. Ce mode présente des risques, puisqu'en cas d'erreur de configuration il est possible de rendre un site inaccessible, mais il apporte aussi des garanties fortes sur l'utilisation de communications sécurisées. Il existe même certaines listes pré-chargées dans les navigateurs pour appliquer ce mode dès la première connexion sur certains sites. Bien entendu, HSTS est spécifique à HTTPS, et difficilement généralisable à d'autres protocoles reposant sur TLS.

Une idée similaire a été proposée pour mettre en œuvre le *Certificate Pinning* à l'échelle d'internet : *HTTP Public Key Pinning* (HPKP [19]). Lorsqu'un serveur présente l'extension dans un canal HTTPS correctement authentifié, il peut indiquer au navigateur les informations à épingler pour les usages futurs. Afin de limiter les risques d'erreur de configuration, le navigateur doit vérifier qu'au moins un des *pin* correspond à la chaîne de certification de la session TLS courante, *et* qu'au moins un *pin* ne lui correspond *pas* (afin d'inciter les administrateurs à disposer de chaînes de certification de secours). De plus, il existe un mode *report-only* dans lequel le navigateur n'applique pas la restriction, mais se contente de rapporter les écarts constatés avec la politique.

Ce mécanisme permet facilement d'imposer des restrictions supplémentaires selon la philosophie TOFU, mais il est également possible de

pré-charger tous les *pins* visibles en parcourant une liste de sites, comme le prévoient Google et Mozilla. HPKP a également été pensé pour permettre une mise en place progressive, avec une phase de test de la politique. Cependant, pour de nombreux sites, la protection supplémentaire n'est offerte qu'après une première connexion fructueuse.

Certificate Transparency *Certificate Transparency* (CT [33]) est un projet poussé par Google pour rendre les certificats émis par une autorité plus visibles. L'idée est en particulier de permettre au propriétaire d'un domaine donné de pouvoir vérifier de manière rapide et robuste quels sont les certificats qui ont été émis pour son nom de domaine.

Pour cela, *Certificate Transparency* utilise des *certificate logs*, des journaux publics, en *append-only*, dont l'intégrité est garantie cryptographiquement (la structure utilisée est un arbre de Merkle [44]). Afin que ces journaux deviennent obligatoires pour la publication des certificats, l'idée est que les navigateurs vérifient la présence d'une preuve que le certificat appartient bien à un journal. Cette preuve peut être transmise à l'aide d'une extension X.509, via une extension TLS ou au travers de l'*OCSP Stapling*. Il devient alors possible de s'assurer qu'un certificat émis est présent dans l'arbre (et donc visible de tous) avant de l'utiliser.

Actuellement, Google exige des autorités de certifications d'implémenter CT pour les certificats EV dans Chrome. Bien que cette solution soit présentée comme *la* solution aux problèmes de certificats sur Internet, la technologie est encore jeune, et peu d'acteurs (en particulier chez les autorités de certification) ont à ce jour accepté de se lancer pleinement dans l'aventure, qui nécessite le déploiement de *logs*, mais aussi de services de vérification pour que la sécurité soit effectivement améliorée. Or c'est l'adoption massive qui peut seule donner du sens à ce mécanisme. Enfin, CT ne sert qu'à la détection, le volet révocation étant un autre projet à venir (*Revocation Transparency*[24]).

Ainsi, face aux problèmes complexes de la détection de certificats non légitimes et de la reprise sur ces incidents, les solutions sont toujours en cours de définition/standardisation. Si certaines idées émises dès 2011 ont rapidement disparu (*Sovereign Keys* de l'EFF[25], *Convergence* de Moxie Marlinspike[26], ou encore *TACK* proposé par Trevor Perrin[27]), le développement et le déploiement avance pour d'autres idées.

24. `http://www.links.org/files/RevocationTransparency.pdf`
25. `https://www.eff.org/sovereign-keys`
26. `http://convergence.io/`
27. `http://tack.io/`

Étant donné le modèle de confiance HTTPS, où la question est la plus délicate à résoudre, une solution concrète devra sans doute reposer sur plusieurs mécanismes complémentaires, à la manière de ce que Mozilla a proposé courant 2014 [28] : des listes de révocation compilées et transmises par mise à jour, l'utilisation de l'extension *Must-Staple*, *HTTP Strict Transport Security* et *HTTP Public Key Pinning*, les certificats à durée de vie courte, etc.

5 2014 : une pluie d'erreurs d'implémentation

5.1 CVE-2014-1266 : `goto fail` Apple

En février 2014, Apple a publié une annonce alarmante, indiquant que certaines versions de ses systèmes d'exploitation avait une faille majeure dans TLS : il était possible de contourner l'authentification des serveurs.

La faille fut vite identifiée : à cause d'une ligne `goto fail` répétée dans le code (voir figure 10), une bonne partie de la fonction `SSLVerifySignedServerKeyExchange` était réduite à du code mort. Ainsi, lorsqu'un client négociait certaines suites cryptographiques (DHE+ECDHE), la signature des paramètres échangés par le serveur n'était pas vraiment vérifiée par le client.

Un attaquant pouvait alors simplement répondre à la place d'un serveur légitime, forcer l'utilisation d'une suite vulnérable, présenter les certificats légitimes, et envoyer un message `ServerKeyExchange` de son choix, puisque la signature n'était pas vérifiée : du point de vue du client, le certificat du serveur était correctement vérifié, mais pas les paramètres Diffie-Hellman.

Ce problème a été corrigé rapidement, mais il a montré que le code en question ne faisait pas l'objet de simples vérifications comme la détection de code mort. Certaines voix se sont alors élevées pour promouvoir de meilleurs outils de compilation ou d'analyse, voire l'utilisation de langages alternatifs au C.

5.2 CVE-2014-0092 : `goto fail` GnuTLS

Quelques jours plus tard, GnuTLS annonce une faille dans le code de vérification des signatures. Cette fois, le problème est (un peu) plus subtil : le problème vient des fonctions `check_if_ca` et `_gnutls_verify_certificate2`, censées renvoyer si un certificat correspond à une autorité pour la première, et le résultat de la vérification d'une

28. `https://wiki.mozilla.org/CA:RevocationPlan`

```
SSLVerifySignedServerKeyExchange( ... )
static OSStatus
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Figure 10. Code vulnérable à l'origine du `goto fail` Apple.

signature. Bien que la documentation de ces fonctions indiquait qu'elles renvoyaient un booléen (c'est-à-dire 0 ou 1), la valeur de retour pouvait prendre trois valeurs :

— si le résultat est 1, la signature est correcte ;
— si le résultat est 0, la signature est erronée ;
— dans certains cas d'erreurs de *parsing*, le résultat était négatif (voir figure 11 par exemple).

Ainsi, certaines fonctions utilisant ces routines internes, telles que `gnutls_x509_crt_check_issuer`, acceptaient non seulement les certificat avec une signature correcte, mais également certains certificats mal formés ! En effet, comme le montre la figure 12, seul le cas `ret == 0` menait à un rejet du certificat.

Les développeurs ont corrigé la logique nécessaire, d'une part en s'assurant que les fonctions appelées renvoient effectivement uniquement `0` ou `1` (comme la documentation l'indiquait) et d'autre part pour que les fonctions appelant ces routines soient plus strictes sur le résultat obtenu.

Il est intéressant de remarquer qu'un *bug* similaire avait été trouvé dans OpenSSL en 2008 (CVE-2008-5077). Là encore, la question des outils et du langage a été soulevée.

```
/*
* Returns only 0 or 1. If 1 it means that the certificate
* was successfuly verified. [...]
*/
static int _gnutls_verify_certificate2( ... )
{
    ...

    result = _gnutls_x509_get_signed_data( ... );
    if (result < 0) {
        gnutls_assert();
        goto cleanup;
    }

    ...

cleanup:
    if (result >= 0 && func)
        func(cert, issuer, NULL, out);
    _gnutls_free_datum(&cert_signed_data);
    _gnutls_free_datum(&cert_signature);

    return result;
}
```

Figure 11. Certaines fonctions (ici, `_gnutls_verify_certificate2`) pouvaient retourner une valeur négative en cas d'erreur de *parsing*, contrairement à ce que disaient les commentaires.

```
    ret = _gnutls_verify_certificate2( ... );
    if (ret == 0) {
        /* if the last certificate in the certificate
         * list is invalid, then the certificate is not
         * trusted.
         */
        gnutls_assert();
        status |= output;
        status |= GNUTLS_CERT_INVALID;
        return status;
    }
```

Figure 12. Appel à la fonction `_gnutls_verify_certificate2` depuis la fonction `gnutls_x509_crt_check_issuer` : si la fonction appelée renvoie `-1`, l'exécution continue et le certificat est accepté.

5.3 CVE-2014-0160 : *Heartbleed*

Le 8 avril 2014, une nouvelle vulnérabilité concernant TLS a été publiée, à l'aide d'un site web avec un logo et « une marque » : *Heartbleed*. La vulnérabilité repose sur une mauvaise implémentation de l'extension TLS *Heartbeat*.

Cette extension ajoute à TLS un moyen d'échanger des messages d'un nouveau type, auxquels une implémentation conforme doit répondre en répétant le message reçu. Un tel mécanisme permet en théorie deux choses :

— en faisant varier la taille des messages envoyés, il est possible de découvrir la taille maximale des paquets sur un canal (*Path MTU Discovery*) ;

— en émettant régulièrement de tels messages, non porteurs d'information applicative, il est possible de maintenir une connexion ouverte (*Keep alive*).

En réalité, ces deux avantages sont surtout pertinents pour DTLS, la version datagramme de TLS, que l'on rencontre parfois au-dessus du protocole UDP. Cependant, l'extension *Heartbeat* a été intégrée au code d'OpenSSL pour DTLS *et* TLS, par défaut... le 31 décembre 2011 [29].

La vulnérabilité en elle-même est terriblement simple : face à un message `HeartbeatRequest` annonçant un contenu plus long que le message envoyé, une implémentation vulnérable va remplir le message `HeartbeatResponse` avec le contenu reçu, puis compléter avec ce qui se trouve en mémoire [30].

L'impact, initialement difficile à préciser, s'est révélé critique : en utilisant cette vulnérabilité sur un serveur, il est possible d'espionner des communications entre *d'autres clients* et le serveur. L'ensemble des données allouées sur le tas peuvent ainsi être récupérées : le contenu déchiffré des communications TLS (dont les cookies et autres mots de passe), mais aussi tout élément interne au service, tel que sa clé privée.

5.4 CVE-2014-0224 : *EarlyCCS*

En juin 2014, une attaque sur la machine a état d'OpenSSL a été publiée par Masashi Kikuchi [31]. Un attaquant situé entre un client *et* un

29. La première version d'OpenSSL officielle contenant *Heartbeat* est la version 1.0.1 du 14 mars 2012.

30. Au passage, il est intéressant de lire la RFC correspondante [52], qui, mise en regard de la spécification de TLS [16], n'est pas limpide sur la conduite à tenir en cas de message incomplet : un tel message peut-il être éclaté sur plusieurs *records* ?

31. `http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html`

serveur OpenSSL, *tous les deux* vulnérables pouvait alors réaliser une attaque de type *man-in-the-middle*, en forçant les deux interlocuteurs à utiliser des clés faibles. Le schéma 13 décrit l'attaque.

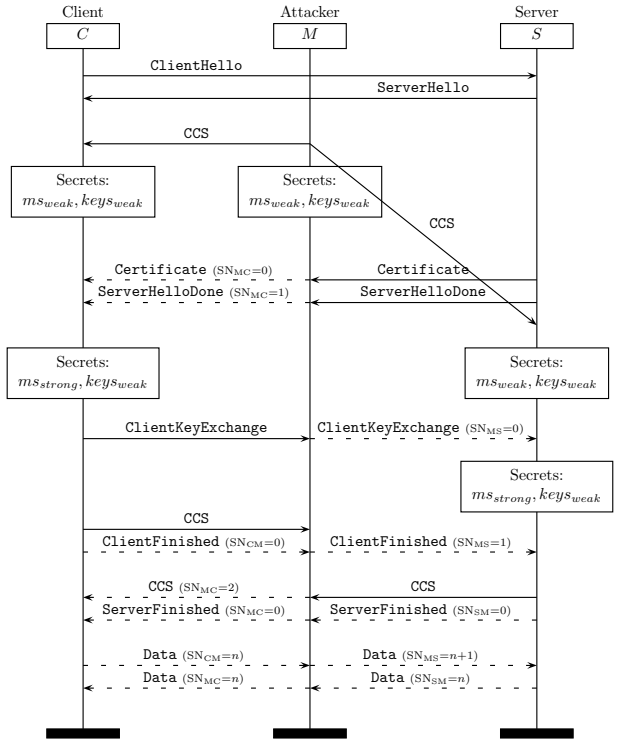


Figure 13. Description de l'attaque *EarlyCCS*. *ms* signifie *master secret* et SN_{XX} désigne le numéro du *record* transmis. Il y a en pratique quatre numéros à retenir : entre le client et l'attaquant et entre l'attaquant et le serveur (à chaque fois un compteur pour chaque sens). Source : équipe INRIA Prosecco.

L'idée essentielle de l'attaque est d'exploiter la machine à état d'OpenSSL qui accepte de recevoir un message `ChangeCipherSpec` un peu plus tôt que prévu (lorsque le second message `ChangeCipherSpec` sera reçu, il sera simplement ignoré), que ce soit dans son comportement client ou serveur. Comme aucun *master secret* n'est défini, les clés sont générées à partir d'un *pre-master secret* nul et des aléas échangés précédemment, qui sont publics. Il s'agit ensuite pour l'attaquant de garder chaque connexion synchronisée comme il se doit, en tenant compte des canaux protégés avec

les clés faibles dans un seul sens, ainsi que des compteurs pour les motifs d'intégrité.

Enfin, pour que la négociation se termine correctement, il faut pouvoir transmettre au client et au serveur un message *Finished* correct. Or ce dernier doit contenir un haché du *master secret* qui a finalement été correctement négocié. C'est la raison pour laquelle il est nécessaire que le client et le serveur soient vulnérables pour pouvoir monter l'attaque.

2015 nous a depuis appris qu'il existait d'autres problèmes liés à la machine à état de diverses piles TLS (voir la section 5.8 plus bas).

5.5 CVE-2014-3511 : OpenSSL *downgrade attack*

En juillet 2014, David Benjamin et Adam Langley ont montré qu'OpenSSL avait une réaction étrange face à un `ClientHello` trop fragmenté. En effet, la spécification TLS autorise à éclater les messages de manière essentiellement arbitraire [32].

Cependant, si OpenSSL ne trouvait pas suffisamment d'information dans le premier fragment d'un `ClientHello`, la version négociée était systématiquement TLS 1.0. En pratique, OpenSSL a en effet besoin d'avoir les 6 premiers octets du `ClientHello` pour déterminer la version proposée par le client (voir figure 14).

Couche	Champ	Taille	Valeur
Record	Type	1	`16` (*Handshake*)
	Version	2	`03 01` (TLS 1.0)
	Longueur	2	longueur du *record*
Handshake	Type	1	`01` (`ClientHello`)
	Longueur	3	longueur du message
`ClientHello`	Version	2	`03 02` (TLS 1.1)
	...	...	...

Figure 14. Contenu des premiers octets d'un `ClientHello` standard. C'est la version portée par la couche *Handshake* qui sert à la négociation.

Il est important de constater qu'un message *Handshake* peut *légitimement* être fragmenté. En effet, la longueur d'un tel message est encodée sur 3 octets, alors qu'un *record* a au maximum une longueur de 16 Ko.

32. En pratique ce qui est autorisé et interdit n'est pas toujours très clair dans les spécifications. Cette spécification peu claire a mené à d'autres attaques (*Alert attack* [9]), et peut être vue comme une des causes de la faille *Heartbleed*.

La version actuelle d'OpenSSL rejette désormais purement et simplement ces `ClientHello` fragmentés, ce qui est incorrect vis-à-vis de la spécification, mais l'alternative est décrite comme trop difficile à implémenter. Cette attaque, tout comme la précédente, montre que la complexité de TLS, combinée avec les besoins de supporter plusieurs versions du protocole, pose de nombreux problèmes aux développeurs.

5.6 CVE-2014-1568 : NSS/CyaSSL/PolarSSL Signature Forgery

En septembre, une nouvelle vulnérabilité client permettant de contourner l'authentification d'un serveur a été publiée, cette fois concernant NSS, la bibliothèque cryptographique de Firefox [33]. La faille se situe ici dans le décodage ASN.1 DER des signatures, et repose sur trois éléments :

— l'exposant public de la clé RSA dont on cherche à imiter la signature doit valoir 3 ;
— le *parser* ASN.1 DER doit être laxiste, et accepter des encodages non canoniques [34] ;
— le traitement de certaines longueurs doit provoquer un débordement d'entier.

Dans TLS, les signatures RSA suivent le standard PKCS#1 v1.5 : pour signer un message m avec la clé RSA privée (N, d) et l'algorithme de hachage H, on suit les étapes suivantes :

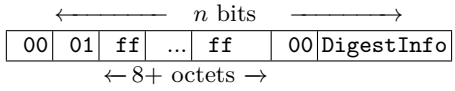
— Hachage : on calcule $h = H(m)$;
— Mise en forme : on prépare un bloc ASN.1 DER consistant en une séquence contenant l'identifiant de la fonction de hachage utilisée et le haché h. Notons d ce bloc, appelé `DigestInfo` ;
— Bourrage (*padding*) : si n est la longueur du module n, on crée un grand entier sur n bits, dont la valeur commence par les octets `00 01 ff ... ff 00`, suivis de d. Soit x l'entier représenté par cette valeur (voir figure 15) ;
— Signature : le résultat de la signature est $x^d[n]$.

Une première attaque avait été proposée en 2006 par Bleichenbacher, pour abuser des implémentations ne vérifiant pas l'absence de données à la suite du bloc `DigestInfo`. Ainsi, dans le cas d'un exposant public égal à 3, il était possible de forger trivialement une signature ayant la forme

33. Un examen plus poussé a montré que d'autres implémentations comme CyaSSL et PolarSSL étaient également touchées par la vulnérabilité.

34. Le DER (*Distinguished Encoding Rules*), est une représentation concrète de la syntaxe abstraite ASN.1 sous forme normale : il existe une et une seule représentation acceptable.

Format PKCS#1 v1.5 :



`DigestInfo` :

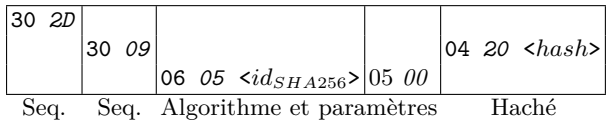


Figure 15. Format PKCS#1 v1.5. *Seq.* est une séquence ASN.1, les valeurs en italiques représentent des longueurs, et `<hash>` est le haché (ici SHA-256) du message à signer.

indiquée à la figure 16. Il suffit alors de remplir les octets réservés en fin de message (le champ *garbage* du schéma) par des zéros : on obtient un message m. Ensuite, on note s la partie entière supérieure de la racine cubique *réelle* de m ; si n est suffisamment grand, s^3 sera égale à m sur la première partie du message, au moins jusqu'à `DigestInfo`.

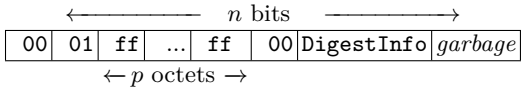


Figure 16. Exemple d'un message PKCS#1 v1.5 mal formaté. En cas de *parsing* laxiste, ce message facilement forgeable est accepté. Certaines implémentations autorisent de plus p a être réduit à 0, ce qui laisse plus de place en fin de message.

La vulnérabilité mise au jour par l'équipe INRIA Prosecco est plus subtile, puisqu'elle repose sur une représentation non canonique du bloc `DigestInfo`. Ainsi, au lieu de représenter la longueur du haché SHA-1 par un simple octet à `0x14`, on peut utiliser une représentation alternative, plus longue, normalement interdite en DER, `8f 00 .. 00 14` [35].

S'il n'y avait que cela, la vulnérabilité serait complexe à exploiter, mais en réalité, les premiers octets encodant la longueur peuvent être arbitraires, car la lecture d'un entier sur plus de 4 octets provoque un

35. Le bit de poids fort du premier octet indique une représentation longue : les bits restants représentent le nombre d'octets utilisés pour encoder la longueur, ici 15.

débordement d'entier : dans l'exemple donné, sur les 15 octets, les 11 premiers peuvent donc servir de variable d'ajustement pour les calculs.

Là encore, pour que les calculs soient réalisables, un exposant public valant 3 est nécessaire en pratique.

Concernant le traitement des signatures PKCS#1 v1.5, le correctif classiquement admis est d'inverser la méthode de vérification : plutôt que de décoder le bloc `DigestInfo` et de vérifier que le haché obtenu est le bon, il est plus sûr de produire le `DigestInfo` attendu (qui est censé être canonique) et de vérifier que les représentations concrètes sont similaires.

5.7 CVE-2014-6321 : Exécution de code arbitraire dans SChannel

En novembre, Microsoft a publié un bulletin d'alerte, MS14-066, comportant plusieurs vulnérabilités. L'une d'elles concernait SChannel, l'implémentation TLS de Microsoft, et pouvait mener à de l'exécution de code arbitraire.

La faille se situait dans la fonction `DecodeSigAndReverse`, responsable de *parser* la signature ECDSA réalisée dans le cas d'une authentification client par certificat. En théorie, seul les serveurs configurés pour accepter les certificats client sont censés envoyer un message `CertificateRequest`, mais SChannel acceptait les messages `Certificate` et `CertificateVerify` des clients, *y compris* lorsque le serveur n'avait rien demandé [36] !

Dans le cas d'un certificat utilisant les courbes elliptiques, il est nécessaire, avant de pouvoir interpréter la signature, de déterminer la courbe sur laquelle se fait la signature. La courbe en question est décrite dans la clé publique qui sert à vérifier la signature, soit sous forme implicite (certaines courbes ont des OID ASN.1 prédéfinis, comme `1.3.132.0.34` pour `secp384r1`), soit de manière explicite (le corps sur laquelle la courbe est construite, ainsi que les paramètres de l'équation). En particulier, cette description donne la taille des coordonnées utilisées. Ensuite, lire la signature revient à extraire l'abscisse et l'ordonnée d'un point de la courbe.

Dans le cas présent, le code alloue des tableaux pour accueillir les coordonnées de la signature, en se basant sur la description de la courbe, puis réalise la lecture des coordonnées depuis la signature en utilisant la longueur des entiers ASN.1, sans vérifier la cohérence avec la taille

36. On retrouvera cette propension à interprété des messages non sollicités dans certaines piles TLS avec FREAK (voir section 5.8

précédente. On peut donc simplement déclencher un débordement en forgeant une fausse signature avec des coordonnées trop longues

Il est donc possible d'écraser une partie du tas en jouant sur les longueurs présentées dans la signature. Des preuves de concept ciblant les serveurs IIS ont montré qu'il était possible de déclencher de l'exécution de code arbitraire sur les systèmes vulnérables.

5.8 Quoi de neuf en 2015 ?

En 2015, plusieurs avis de sécurité concernant TLS ont déjà été publiés :
— le 8 janvier, OpenSSL a publié plusieurs correctifs relatifs à des erreurs de gestion mémoire classiques ;
— le même avis de sécurité comportait également des problèmes liés à la gestion de l'automate d'état TLS dans OpenSSL n'est pas robuste (tout comme l'attaque *Early CCS* l'avait montré) ;
— le 14 janvier, PolarSSL a annoncé que le *parser* ASN.1 souffrait de plusieurs erreurs d'implémentation pouvant mener à de l'exécution de code arbitraire.

Attardons-nous sur le second point, dont on a appris plus de détails en mars dernier, avec la publication du site SMACK[37] (*State Machine AttaCKs*). À l'aide de FlexTLS, un pile TLS flexible développée par l'équipe INRIA Prosecco, des chercheurs ont testé les machines à état de nombreuses piles TLS [6]. Les résultats sont impressionnants, et touchent quasiment toutes les piles TLS connues. Voici les principales attaques découvertes, qui reposent toutes sur un attaquant réseau actif.

***Early Finished* (usurpation de l'identité du serveur)** L'attaquant répond à la place du serveur avec les messages suivants : `ServerHello`, `Certificate` (avec le certificat du serveur à usurper) et `ServerFinished`, en omettant le reste de la négociation. Face à une telle négociation abrégée, les implémentation JSSE (Java) et CyaSSL considèrent le serveur authentifié, et transmettent les messages `ApplicationData` en clair !

***Skip Verify* (usurpation de l'identité du client)** Dans le cas d'une connexion avec authentification mutuelle, le serveur demande au client de présenter un certificat (message `Certificate`) et de signer avec sa clé privée une partie des messages *Handshake* (message `CertificateVerify`) avant de considérer la connexion comme authentifiée. Cependant certaines implémentations acceptent de recevoir un certificat client sans le

37. `https://www.smacktls.com/`

message `CertificateVerify` : l'implémentation TLS de Mono considère simplement ce second message comme optionnel, et considère tout de même le client authentifié ; avec CyaSSL, il faut également omettre le message `ChangeCipherSpec` du client ; avec OpenSSL, la faille est plus subtile, et nécessite que l'attaquant présente un certificat client avec une clé Diffie-Hellman statique.

***Skip ServerKeyExchange* (perte de la *forward secrecy*)** On suppose qu'un client et un serveur négocient une suite utilisant ECDHE pour l'échange de clé et ECDSA pour l'authentification du serveur. Avec certaines implémentations, si un attaquant supprime le message `ServerKeyExchange`, le client utilisera la clé publique présente dans le certificat ECDSA à la place de la part de secret ECDHE attendue. Cependant, les transcripts étant différents côté client et serveur, la connexion échouera lors de l'échange des messages `Finished`. L'attaque peut avoir un impact lorsque *False Start* (l'envoi de données applicatives avant que la négociation soit terminée) est utilisé. OpenSSL et NSS sont concernés par cette vulnérabilité.

FREAK (*Factoring RSA Export Keys*) La dernière attaque présentée dans l'article est celle qui a fait couler le plus d'encre : FREAK. Comme les attaques précédentes, elle repose sur un attaquant réseau actif qui modifie les messages à la volée.

La section 1 présentait une connexion TLS classique, reposant sur l'échange de clé par chiffrement RSA. Jusqu'au début des années 2000, l'utilisation de mécanismes de chiffrement était limité à l'export par les États-Unis, et pour être compatible avec cette réglementation, il existait un autre mécanisme d'échange de clé, RSA-EXPORT : au lieu de transmettre simplement son certificat, le serveur transmettait son certificat (contenant une clé RSA long terme de 1024 ou 2048 bits), puis une clé publique RSA de petite taille (512 bits), signée par la clé long terme [38]. Le client chiffrait alors son *pre-master secret* avec la clé de petite taille. Cela permettait d'obtenir une authentification du serveur avec une clé de grande taille, tout en restant compatible avec les règles concernant le chiffrement.

Le problème soulevé par FREAK est triple :

— certains clients ayant négocié le chiffrement RSA standard acceptent de recevoir un message `ServerKeyExchange` contenant une clé

38. Pour éviter le rejeu, cette signature couvre les aléas client et serveur échangés au début de la connexion.

RSA export de 512 bits, pourtant réservé à l'échange de clé RSA-EXPORT ;
— de (trop) nombreux serveurs TLS acceptent de négocier des suites EXPORT (des statistiques[39] annonçaient début mars que 35 % des serveurs HTTPS avec un certificat reconnu par les navigateurs étaient dans ce cas !) ;
— parmi ces serveurs, beaucoup réutilisent la même clé RSA-EXPORT pendant toute la durée de vie du serveur, ce qui rend possible une attaque en pratique.

Le scénario est le suivant :
— A trouve un serveur S acceptant des suites EXPORT avec une clé RSA 512 persistante ;
— A casse la clé RSA 512 (en quelques heures avec un budget raisonnable) ;
— A se place entre un client C vulnérable et le serveur S pour que le client négocie un échange de clé RSA et que le serveur négocie un échange de clé RSA-EXPORT ;
— A transmet les messages de part et d'autre, en déchiffrant au passage le *pre-master secret* chiffré par le client avec la clé faible ;
— enfin, A réécrit les message `Finished` (ce qui est possible puisqu'il dispose du secret de session).

FREAK a fait du bruit car les piles OpenSSL, BoringSSL, LibReSSL, SecureTransport (Apple), SChannel (Microsoft), Mono et IBM JSSE (Java) se sont révélées vulnérables.

6 TLS 1.3 : un nouvel espoir ?

Avant de présenter TLS 1.3, il est utile de tirer des enseignements de l'ensemble des failles présentées. Certains aspects peuvent être traités à l'aide d'une nouvelle version du standard, mais pas tous.

6.1 Enseignements tirés

Inclure l'échec dans les propriétés à tester Il est naturel dans un développement logiciel de vérifier que le produit fini remplit le cahier des charges, c'est-à-dire que ce qui doit fonctionner fonctionne. Il est beaucoup plus rare de tester que ce qui doit échouer échoue effectivement. En effet, cette démarche est une démarche de sécurité, et non une démarche fonctionnelle.

39. `https://freakattack.com/`

Force est de constater qu'aucun (ou presque) test négatif n'est réalisé en pratique sur les piles TLS, puisque des erreurs triviales sont régulièrement trouvées sur les produits : absence de vérification de l'extension *BasicConstraints* dans les certificats, contournements du code de vérification dans certains cas, etc. La situation est même pire, puisque des problèmes connus dans une pile TLS à un instant donné peuvent réapparaître dans une autre pile 5 ou 10 ans plus tard [40] : aucune base de tests publique ne permet de vérifier proprement la non-régression !

Quelques efforts sont en cours dans la communauté scientifique, mais la meilleure base de certificats publics existante à ce jour est *Frankencerts* [13], un ensemble de certificats forgés de bric et de broc à partir de certificats valides. L'idée était de créer des certificats syntaxiquement corrects, mais dont la sémantique pouvait être invalide. Ces travaux sont un premier pas, mais ont plusieurs limitations : ils ne touchent pas à la structure ASN.1 DER des certificats ; de plus, la méthodologie proposée par les chercheurs laisse à désirer, puisqu'ils considèrent que la réponse attendue pour un certificat est ce que la majorité des piles TLS répondent.

Améliorer la qualité du développement logiciel En plus de tests en boîte noire avec une liste de problèmes connus, il est également essentiel d'améliorer la qualité du code dans les piles TLS. En effet, ces briques logicielles représentent un élément crucial dans la sécurité des systèmes d'information. Il est donc important d'une part de s'assurer qu'elles remplissent leur rôle, et d'autre part de vérifier qu'elles n'affaiblissent pas la sécurité des systèmes qui les embarquent. En effet, au moins deux failles en 2014 ont eu un impact qui allait au-delà de la remise en cause de TLS, permettant la divulgation d'information échangées par TLS à des tiers (*Heartbleed*, section 5.3) ou l'exécution de code arbitraire (faille dans SChannel, section 5.7).

Pour améliorer la qualité de ces composants, il faut mettre à profit les outils d'analyse existants, tels que ceux offerts par les compilateurs. Cela implique également de rendre le code auditable. Enfin, des outils de *fuzzing* permettent d'instrumenter le code pour secouer les implémentations de l'intérieur ; c'est le cas d'`afl` (*American Fuzzy Lop*, développé par Michal

40. Par exemple, Moxie Marlinspike a redécouvert la même vulnérabilité sur la non-vérification de l'extension *BasicConstraints* des certificats à 10 ans d'intervalles, dans les implémentations Microsoft en 2002 (CVE-2002-0862 et CVE-2002-1183) et Apple en 2011 (CVE-2011-0228) ; de même, une faille similaire à celle découverte en 2014 sur GnuTLS (voir section 5.2, CVE-2014-0092) avait été trouvée sur OpenSSL en 2008 (CVE-2008-5077).

Zalewski) qui a par exemple permis d'identifier un débordement de tampon dans le tas dans GnuTLS.

Une voie alternative est d'utiliser d'autres langages de programmation que le C pour garantir certaines propriétés de sécurité de manière intrinsèque. Les chercheurs de Cambridge développant Mirage ont ainsi publié une implémentation complète de TLS en OCaml [41]. Si cette démarche a des vertus, il ne faut pas moins étudier la sécurité des bibliothèques et programmes résultants. En effet, si OCaml garantit par construction l'absence de certaines classes d'erreur *sous condition* [42], et si l'utilisation du *pattern matching* exhaustif peut éviter certaines erreurs de logique, TLS reste intrinsèquement complexe, et des failles peuvent également apparaître dans couches haut niveau (interprétation de l'ASN.1, machine à état, etc.). La validation de ces implémentations passent là encore par du test et de l'audit.

TLS est complexe L'encodage ASN.1 DER est relativement complexe, comme le démontrent les nombreuses failles de sécurité des *parsers* de certificats. Comme l'authentification des serveurs repose uniquement sur des certificats X.509 [43], cette part de complexité restera une source de problèmes pour TLS pendant longtemps. Il est donc nécessaire de disposer d'outils robustes et éprouvés.

TLS contient encore de nombreux algorithmes et mode cryptographiques historiques, dont on sait qu'ils sont dangereux, par exemple le paradigme *MAC-then-Encrypt* ou le chiffrement RSA PKCS#1 v1.5. Ils sont intrinsèquement difficiles à implémenter correctement, mais TLS 1.3 prévoit de retirer la majorité des suites cryptographiques présentant ces défauts.

Enfin, une autre source de complexité de TLS est son automate d'état. Comme l'ont montré des attaques comme la vulnérabilité sur la renégociation en 2009, *Triple Handshake* et *EarlyCCS* en 2014, ou plus récemment l'acceptation de séquences de messages incorrectes dans OpenSSL, il est difficile d'analyser le fonctionnement réel de l'automate idéal ou des implémentations concrètes. Sur ce point, des travaux récents menés par l'équipe

41. On peut aussi citer miTLS, implémentation prouvée en F#, mais l'objectif de miTLS est surtout de fournir une implémentation de référence, et un socle pour prouver des propriétés de sécurité sur le protocole.

42. L'utilisation classique d'OCaml évite en effet les débordements de tableau, mais pas si certaines fonctions marquées `unsafe` sont employées, ou si du code C est intégré, ce qui est le cas du projet Mirage.

43. Il existe en théorie des moyens d'authentification alternatifs, mais ils représentent une part infime des usages TLS.

INRIA Prosecco apportent des éléments de réponses, que ce soit sur la spécification (miTLS est une implémentation sur laquelle des propriétés de sécurité ont été prouvées, sur des situations réelles) ou sur le test des implémentations (FlexTLS est une pile TLS flexible permettant de secouer les automates d'état). De plus, TLS 1.3 devrait retirer des options complexes telles que la renégociation ou la reprise de session, pour obtenir un automate d'état beaucoup plus simple.

Cependant, si TLS 1.3 apporte (enfin) des proposition concrètes pour réduire la complexité du protocole, il faut se rappeler qu'une transition est nécessaire, et que les implémentations courantes continueront encore de supporter TLS 1.0 à TLS 1.2 pendant encore des années. En particulier, SSLv3 est encore partiellement supporté aujourd'hui, près de 20 ans après sa spécification, et malgré des failles de sécurité de plus en plus prégnantes. Il est donc déjà temps, lorsque SSLv3 sera enfin abandonné, de mettre effectivement TLS 1.0 et TLS 1.1 sur le banc de touche.

6.2 TLS 1.3

Les premiers échanges concernant TLS 1.3 remontent à novembre 2013, quand Eric Rescorla propose un nouveau mécanisme d'échange de clés pour le protocole [46]. Après quelques mois de discussions sur la liste du groupe de travail TLS de l'IETF, le travail a également continué, à partir d'avril 2014, sur GitHub[44].

Avertissement : cette section a été rédigée en avril 2015, alors que la spécification TLS 1.3 n'est pas encore figée. Les éléments présentés ici reposent sur la version 5 du brouillon[45] *et sur les échanges du groupe de travail TLS de l'IETF.*

Un échange de clé repensé La plus grosse modification du protocole dans TLS 1.3 est sans doute une refonte des messages échangés. La figure 17 présente ce que sera une connexion TLS typique (sans authentification client). Les principales modifications par rapport à TLS 1.2 sont :

— seuls les échanges de clés reposant sur DHE ou ECDHE sont supportés, excluant en pratique le chiffrement RSA et assurant la *forward secrecy*[46] ;

44. `https://github.com/tlswg/tls13-spec`

45. `https://tools.ietf.org/html/draft-ietf-tls-tls13-05`

46. Un support permettant l'utilisation PSK ou SRP sans *forward secrecy* pourrait néanmoins être ajouté.

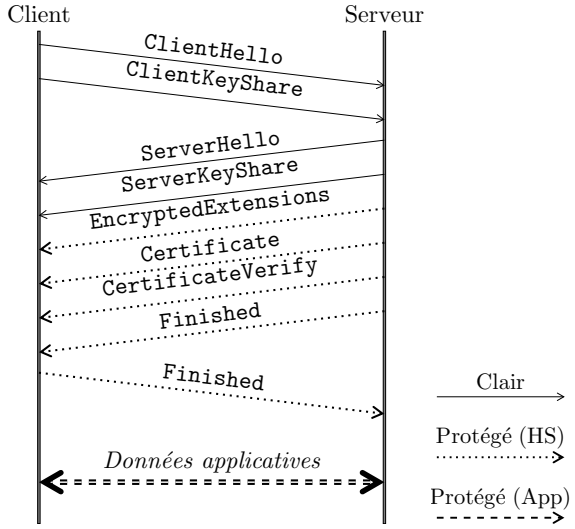


Figure 17. Exemple de négociation avec TLS 1.3.

— les groupes utilisés pour l'échange de clé (les corps finis pour DHE ou les courbes elliptiques pour ECDHE) ne peuvent plus être définis de manière arbitraire par le serveur, mais doivent faire partie d'une liste de groupes prédéfinis [47] ;
— les messages `ClientKeyExchange` et `ServerKeyExchange` ont été renommés en `ClientKeyShare` et `ServerKeyShare` et ont désormais le sens suivant : le client peut proposer un échange de clé pour plusieurs groupes prédéfinis. Puis le serveur choisit un groupe et émet sa part Diffie-Hellman pour le groupe en question. Si tout se passe bien, on a donc une négociation abrégée par rapport à TLS 1.2, puisque l'on n'a besoin que d'1 RTT (*Round-Trip Time*) pour établir la session [48] ;
— si le client n'a proposé des parts de secrets que pour des groupes non supportés par le serveur, ce dernier envoie un message `HelloRetryRequest` dans lequel il impose le groupe à utiliser. On

47. La liste des corps finis a été définie dans un document en cours de standardisation [25] dont l'objet est de restreindre dans TLS 1.0 à TLS 1.2 les corps finis utilisés pour un échange de clé DHE. Pour les courbes elliptiques, les groupes définis par les versions précédentes ont été utilisés, en retirant les courbes elliptiques *explicites*.

48. Pour rester compatible avec les versions précédentes de TLS, le message `ClientKeyShare` peut être embarqué dans une nouvelle extension, `EarlyData`, du `ClientHello`.

se retrouve alors dans une situation analogue à TLS 1.2 (c'est-à-dire 2 RTT) ;
— l'authentification du serveur se fait désormais avec un message `CertificateVerify` similaire à celui de l'authentification client de TLS 1.2.

Afin de simplifier l'automate à état du protocole, les messages `ChangeCipherSpec` ont disparu. Parmi les éléments encore en discussion, on peut citer :
— l'ajout d'un mode 0-RTT, dans lequel le client peut émettre dès le début des données applicatives. La difficulté est qu'il semble impossible d'avoir à la fois une protection anti-rejeu et une distribution fiable de ces données. Pour cette raison, d'âpres discussions ont lieu sur la liste de diffusion...
— la renégociation a été retirée de la spécification, mais les deux cas d'usage légitime de ce mécanisme restent à couvrir : le rafraîchissement des clés et l'authentification client tardive. Si la création d'un message `Update` pour le premier point semble une solution acceptable, le second point n'a pas encore été tranché.

Du nouveau dans la dérivation des clés La dérivation des clés a été profondément modifiée dans TLS 1.3, puisque le *pre-master secret* sert désormais à produire un *handshake master secret*, qui protégera uniquement les messages jusqu'à la fin de la négociation. À la fin de la négociation (c'est-à-dire après authentification optionnelle des parties), deux nouveaux secrets sont dérivés à partir du *handshake master secret* : le *master secret* pour le reste du trafic et le *resumption pre-master secret* qui sera utilisé pour une reprise de session. Grâce à cette distinction, on peut décorréler le cache de session du secret utilisé pour la session courante, ce qui offre une forme de *forward secrecy*.

La fonction de dérivation, la PRF, a également changé, puisqu'elle inclut désormais le haché de l'ensemble des messages de négociation précédents (*session-hash*), et pas uniquement les aléas. Il s'agit de l'inclusion de la contre-mesure à l'attaque *Triple Handshake* [7]. De plus, l'apparition de clés *temporaires* avec le *handshake master secret* permet de protéger une partie des messages de la négociation, notamment certaines extensions envoyées par le serveur dans le message `EncryptedExtensions` et le certificat du serveur.

Des discussions sont en cours pour changer la fonction de dérivation des clés, qui est essentiellement un HMAC aujourd'hui, pour HKDF (*HMAC-based Key Derivation Function*), une primitive plus robuste. De

plus, si le mode 0-RTT était accepté, il faudrait repenser tout l'arbre de dérivation des clés. Ces deux propositions sont actuellement portées par Hugo Krawczyk devant le groupe de travail.

Autres changements Pour terminer, voici quelques autres modifications intéressantes apportées par la nouvelle version de TLS :

- retrait des modes historiques de protection du *Record Protocol* (*streamcipher* et *blockcipher*) : seul le mode AEAD, introduit avec TLS 1.2, est conservé ;
- en particulier, les modes (CBC) et algorithmes (DES, RC4) faibles disparaissent ;
- avec le retrait du chiffrement RSA, le standard de *chiffrement* PKCS#1 v1.5, obsolète, n'est plus employé [49] ;
- la compression a disparu ;
- les aléas envoyés par le client et le serveur ne doivent plus contenir un *timestamp* Unix ;
- les signatures utilisées par le protocole incluent désormais une chaîne de caractères de contexte, afin d'éviter qu'une signature donnée soit réutilisée dans une autre session ou un contexte différent.

TLS 1.3 est un standard en cours de spécification. Un grand ménage a déjà été réalisé, mais certaines fonctionnalités, telles que le mode 0-RTT ou l'authentification tardive du client, donnent actuellement un nouveau tournant aux échanges. En effet, les solutions proposées sont parfois complexes et risquent de faire retomber TLS dans ses anciens travers. Une solution simple sera de ne pas utiliser ces fonctionnalités, mais elles seront certainement implémentées (voire activées par défaut) dans toutes les piles logicielles courantes...

Conclusion

Depuis 2012, de nombreuses attaques ont été présentées sur SSL/TLS, qu'il s'agisse de failles protocolaires, de vulnérabilités cryptographiques, ou d'erreurs d'implémentation. L'actualité a montré qu'aucune pile TLS couramment employée n'avait été épargnée.

Cela ne signifie pas que TLS doit être abandonné, mais cela milite en faveur de la mise en place de la défense en profondeur : TLS ne peut

49. En revanche, la signature PKCS#1 v1.5, qui pose aussi certains problèmes, reste présente dans les messages *CertificateVerify*, et dans les signatures de certificats, dès que RSA est utilisé.

pas être le seul mécanisme de sécurité, et le protocole doit être employé sur des machines à jour, durcies, et le périmètre de tous les composants logiciels utilisés doit être réduit.

Il est également important de suivre quelques recommandations concernant les versions du protocole utilisées (bannir SSLv2 et SSLv3, et supporter TLS 1.2) et les suites cryptographiques (interdire les suites cryptographiques obsolètes, privilégier les algorithmes récents reposant sur des modes cryptographiques modernes tels que GCM). Il est également important de bien comprendre le modèle de confiance de TLS : l'IGC mondiale autour d'HTTPS comporte de nombreux acteurs, et tous les cas d'emploi ne nécessitent pas forcément de leur faire une confiance absolue. Certains de ces éléments commencent à être précisés dans les documents de bonnes pratiques édités par le groupe de travail UTA (*Using TLS in Application*) de l'IETF.

Afin de vérifier que ces recommandations sont bien comprises et appliquées, il est essentiel de tester ses serveurs. Il est étonnant de découvrir en 2015 qu'un tiers des serveurs HTTPS dans le monde accepte les suites EXPORT !

De plus, de nouveaux travaux sont nécessaires pour améliorer la sécurité des implémentations et celle du protocole. Pour les implémentations, cela passe par plus de tests des propriétés de sécurité (y compris pour vérifier qu'un test négatif échoue bien en pratique) et par une meilleure utilisation des langages de programmation et des outils de développement. Pour le protocole, des efforts sont en cours pour analyser TLS dans sa complexité, et TLS 1.3, qui sera bientôt publié, devrait réduire les fonctionnalités de ce protocole tentaculaire pour se concentrer sur un protocole plus simple, sans ajouter, espérons-le, trop de nouvelles fonctionnalités.

Références

1. N. J. AlFardan, D. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the security of RC4 in TLS and WPA. In *USENIX Security*, 2013.
2. N. J. AlFardan and K. G. Paterson. Lucky Thirteen : Breaking the TLS and DTLS Record Protocols. In *IEEE SSP*, 2013.
3. ANSSI. RGS Annexe B1 : Mécanismes cryptographiques, règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques, version 1.20, 2010.
4. R. Barnes, M. Thomson, A. Pironti, and A. Langley. Deprecating Secure Sockets Layer Version 3.0, December 2014.
5. A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), December 2011.

6. Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean-Karim Zinzindohoue. A Messy State of the Union : Taming the Composite State Machines of TLS. In *IEEE Symposium on Security & Privacy (Oakland)*, 2015.
7. K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension, 2014.
8. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, , Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters : Breaking and fixing authentication over tls. In *IEEE Symposium on Security & Privacy*, 2014.
9. Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with Verified Cryptographic Security. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 445–462, 2013.
10. S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 3546 (Proposed Standard), June 2003. Obsoleted by RFC 4366.
11. S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), April 2006. Obsoleted by RFCs 5246, 6066, updated by RFC 5746.
12. Daniel Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *CRYPTO*, 1998.
13. Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 114–129, Washington, DC, USA, 2014. IEEE Computer Society.
14. D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.
15. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176.
16. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.
17. T. Duong and J. Rizzo. BEAST : Surprising crypto attack against HTTPS. Ekoparty, 2011.
18. D. Eastlake 3rd. Transport Layer Security (TLS) Extensions : Extension Definitions. RFC 6066 (Proposed Standard), January 2011.
19. C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), April 2015.
20. Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android : An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

21. S. Fluhrer and D. McGrew. Statistical Analysis of the Alleged RC4 Keystream Generator. In *FSE*, 2000.

22. Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol Version 3.0, 1996.

23. C. Garman, K. G. Paterson, and T. van der Merwe. Attacks Only Get Better : Password Recovery Attacks Against RC4 in TLS, 2015.

24. Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world : Validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

25. D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS, 2015.

26. P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard), September 2014.

27. P. Hallam-Baker. X.509v3 TLS Feature Extension, December 2014.

28. Kipp E.B. Hickman. The SSL Protocol, 1994-1995.

29. J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), November 2012.

30. P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol : TLSA. RFC 6698 (Proposed Standard), August 2012. Updated by RFC 7218.

31. T. Isobe, T. Ohigashi, Y. Waatanabe, and M. Morii. Full Plaintext Recovery Attack on Broadcast RC4. In *FSE*, 2013.

32. Itsik Mantin. Bar-Mitzva Attack : Breaking SSL with 13-Year Old RC4 Weakness. Black Hat Asia, 2015.

33. B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013.

34. Arjen Lenstra, Xiaoyun Wang, and Benne de Weger. Colliding X.509 certificates. In *Cryptology ePrint Archive, Report 2005/067*, 2005.

35. Olivier Levillain. SSL/TLS : état des lieux et recommandations. In *SSTIC*, 2012.

36. Olivier Levillain, Baptiste Gourdin, and Hervé Debar. TLS Record Protocol : Security Analysis and Defense-in-Depth Countermeasures for HTTPS. In *ASIACCS*, 2015.

37. Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the tls protocol. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 62–72, New York, NY, USA, 2012. ACM.

38. B. Moeller and A. Langley. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. RFC 7507 (Proposed Standard), April 2015.

39. B. Möller. Security of CBC Ciphersuites in SSL/TLS : Problems and Countermeasures, 2002-2004.

40. B. Möller, T. Duong, and K. Kotowicz. Google Security Advisory : This POODLE Bites : Exploiting The SSL 3.0 Fallback, 2014.
41. K. G. Paterson and N. J. AlFardan. Plaintext-Recovery Attacks Against Datagram tls. In *NDSS*, 2012.
42. A. Popov. Prohibiting RC4 Cipher Suites. RFC 7465 (Proposed Standard), February 2015.
43. A. Prado, N. Harris, and Y. Gluck. SSL, Gone in 30 seconds - A BREACH beyond CRIME. Black Hat USA, 2013.
44. Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, pages 369–378, 1987.
45. M. Ray. Authentication Gap in TLS Renegotiation, 2009.
46. E. Rescorla. New Handshake Flows for TLS 1.3, 2013.
47. E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012.
48. E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.
49. J. Rizzo and T. Duong. The CRIME attack. Ekoparty, 2012.
50. P. Rogaway. Problems with proposed IP Cryptography. www.cs.ucdavis.edu/r̃ogaway/papers/draft-rogaway-ipsec-comments-00.txt, 1995.
51. S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard), June 2013.
52. R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520 (Proposed Standard), February 2012.
53. Alex Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David A Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today : Creating a rogue CA certificate, 2008. 25th Chaos Communications Congress, Berlin, Germany.
54. A. Stubblefield, J. Ioannidis, and A. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. In *NDSS*, 2002.
55. A. Shulman T. Be'ery. A Perfect CRIME ? TIME Will Tell. Black Hat EU, 2013.
56. S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), March 2011.
57. S. Vaudenay. Security Flaws Induced by CBC Padding Applications to SSL, IPsec, WTLS. In *Eurocrypt*, 2002.
58. David Wagner and Bruce Schneier. Analysis of the SSL 3.0 Protocol. In *Unix Workshop on Electronic Commerce*, pages 29–40. USENIX Association, 1996.
59. Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *Eurocypt'05, LNCS 3494*, pages 19–35, 2005.
60. T. Zoller. TLS/SSLv3 renegotiation vulnerability explained, 2009-2011.