SSL/TLS: état des lieux et recommandations

Olivier Levillain

6 juin 2012 (Conférence SSTIC)

Résumé

SSL/TLS est un protocole ayant pour but de créer un canal de communication authentifié, protégé en confidentialité et en intégrité. L'objectif initial de SSL/TLS était la sécurisation du protocole HTTP, mais son champ d'application s'est élargi depuis : protection d'autres services comme SMTP ou LDAP, création de réseaux privés virtuels (VPN), sécurisation de réseaux sans-fil (EAP-TLS).

Après un rapide historique des différentes versions de SSL/TLS et une description du protocole, cet article présente un panorama des attaques connues, qu'elles portent sur le protocole en lui-même, les algorithmes cryptographiques, ou les certificats mis en œuvre.

Cette énumération des vulnérabilités de SSL/TLS permettra de déduire des recommandations pour une utilisation sécurisée de SSL/TLS. Chaque recommandation sera validée par une expérimentation visant à déterminer la faisabilité de leur application.

Table des matières

Introduction

SSL (*Secure Sockets Layer*) et TLS (*Transport Layer Security*) sont deux variantes d'un même protocole. Leur objectif est de fournir un certain nombre de services pour sécuriser un canal de communication :
— authentification unilatérale ou mutuelle ;
— confidentialité des données échangées de bout en bout ;
— intégrité des données 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) [1].

La section 1 de ce document présente l'historique et le fonctionnement de SSL/TLS. La section 2 s'attache aux vulnérabilités connues des différentes versions du protocole, afin de justifier les versions de SSL/TLS qu'il est recommandé d'employer. Ensuite, la section 3 détaille les algorithmes cryptographiques qui peuvent être négociés par la couche SSL/TLS et met en évidence les suites à utiliser. La section 4 propose un panorama des failles concernant les certificats utilisés dans SSL/TLS. Chacune de ces trois sections formule des recommandations, indique comment les mettre en œuvre et les tester, et évalue le côté réaliste des recommandations vis-à-vis des principaux outils existants.

1 SSL/TLS : un peu de contexte

1.1 Historique

1.1.1 Netscape et l'origine de SSL

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 2.0 [16], rendue disponible en 1995. Elle permettait l'emploi dans Netscape Navigator d'URL de la forme `https://site.domaine.tld`. HTTPS utilise des connexions TCP sur le port 443 par défaut.

SSLv2 fut rapidement suivi d'une version 3.0 [15], qui corrige des failles conceptuelles importantes (voir en particulier la section 2.1). Bien qu'il existe un mode de fonctionnement de compatibilité, les messages de la version 3.0 diffèrent de ceux de la version 2.0.

1.1.2 La normalisation par l'IETF : TLS

En 2001, SSL a fait l'objet d'une normalisation par l'IETF (*Internet Engineering Task Force*) appelée TLS (*Transport Layer Security*). Fondamentalement, TLSv1.0 [10] peut être considéré comme une version 3.1 de SSL. C'est d'ailleurs, le numéro de version utilisé dans les échanges réseau du protocole pour désigner TLSv1.0. Contrairement au passage de SSLv2 à SSLv3, TLS n'a pas été l'objet de changements structurels. On peut cependant noter que quelques modifications dans l'utilisation des fonctions de hachage rendent les messages TLSv1.0 et SSLv3 incompatibles entre eux.

TLSv1.0 fut l'occasion de corriger une faille cryptographique (voir section 2.2) et de mettre en avant des algorithmes cryptographiques libres de droit concernant l'échange de clé et l'authentification : Diffie-Hellman et DSA. L'objectif principal était de proposer une alternative à RSA. Cependant, la publication de la norme TLSv1.0 fut assez longue et intervint en 2001, un an après que RSA fut tombé dans le domaine public.

1.1.3 Versions de TLS

Depuis 2001, TLS a connu quelques évolutions. Dès 2003, un cadre permettant des extensions dans le protocole TLS a été décrit [6]. Cette RFC a été réactualisée en 2006 et 2011 [7, 13]. Comme nous le verrons plus loin (section 2.3), ces extensions permettent de faire évoluer le standard de façon souple, mais requièrent l'abandon de la compatibilité avec SSLv2 et SSLv3.

TLSv1.1 est une révision publiée en 2006 [11] permettant de répondre correctement à certaines attaques cryptographiques mises en évidence sur l'emploi du mode de chiffrement CBC (voir section 3.2.2).

Enfin, en août 2008, une nouvelle version de la norme, TLSv1.2, a été publiée [12]. Il s'agit essentiellement de l'intégration d'éléments épars au sein de la norme. Cette RFC décrit par exemple les extensions TLS comme étant un élément à part entière du standard.

1. 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. Une grande partie de cet article peut s'appliquer à DTLS comme à TLS.

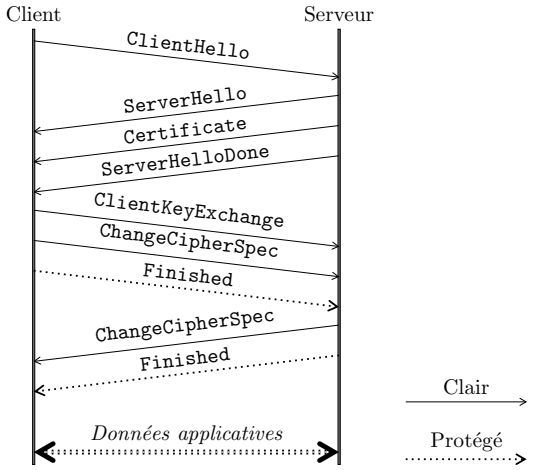

FIGURE 1 – Exemple de négociation TLS avec la suite `TLS_RSA_WITH_RC4_128_MD5`.

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_RC4_128_MD5`, ce qui est assez classique.

1.2.1 `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 quatre fonctions suivantes :
— établir les éléments secrets de session ;
— authentifier les parties ;
— chiffrer les données applicatives ;
— protéger les données applicatives en intégrité.

Ces suites sont définies dans les documents de spécifications des différentes normes, et ont parfois fait l'objet de documents supplémentaires, dans le but d'être intégrées dans une version ultérieure de TLS. Tout comme certains paramètres liés au protocole TLS, ces suites font l'objet d'un registre sur le site de l'IANA [17].

La commande `openssl ciphers -v` permet de lister les suites cryptographiques connues d'OpenSSL. Les quatre mécanismes décrits ci-dessus apparaissent respectivement dans les colonnes `Kx`, `Au`, `Enc`, `Mac`.

Ce message contient d'autres paramètres qui doivent être négociés : la version du standard utilisée (SSLv3, TLSv1, TLSv1.1 ou TLSv1.2), et le mécanisme de compression éventuellement appliqué sur les données applicatives. Enfin, il comporte un champ `ClientRandom`, un aléa fourni par le client qui sera utilisé pour la dérivation des clés.

1.2.2 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` ;
— 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 `ServerRandom`.

Les RFCs décrivant TLS [12, 10, 11] ne définissent pas précisément comment le serveur doit choisir la suite cryptographique. On constate généralement deux comportements possibles :
— le serveur parcourt la liste fournie par le client et choisit la première suite qu'il supporte ;
— le serveur parcourt sa liste de préférences et prend la première suite qui se trouve dans la liste du client.

Par défaut, IIS a le second comportement, et retournera ainsi souvent `TLS_RSA_WITH_RC4_128_MD5`, sa suite préférée. Apache a par défaut le premier comportement, mais il est possible de lui demander d'imposer ses préférences avec la directive `SSLHonorCipherOrder`.

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, le serveur choisit `TLS_RSA_WITH_RC4_128_MD5` :

— `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 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 ;

— `RC4_128` indique que l'algorithme de chiffrement par flot RC4 va être utilisé avec une clé de 128 bits pour chiffrer le canal de communication ;

— `MD5` concerne enfin la protection en intégrité du canal de communication : HMAC MD5 sera employé.

1.2.3 Fin de la négociation

Une fois la suite choisie et le certificat reçu, le client vérifie la chaîne de certification (voir section 4.1). 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 contenant le *pre master secret* chiffré : le `ClientKeyExchange`.

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 `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 (avec RC4 dans notre exemple) et en intégrité (avec HMAC MD5).

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.

1.3 Variantes possibles dans la négociation

L'exemple présenté (`TLS_RSA_WITH_RC4_128_MD5`) est souvent rencontré, mais il existe de nombreuses autres suites cryptographiques possibles. Citons quelques variantes.

1.3.1 Diffie-Hellman éphémère

Un défaut du mécanisme d'échange de clé par chiffrement RSA est qu'il ne protège pas les communications passées contre une divulgation de la clé privée du serveur. En effet, il est facile de se convaincre qu'une fois la clé privée récupérée, il suffit à un attaquant de s'en servir pour déchiffrer le message `ClientKeyExchange`, récupérer le *pre master secret* et obtenir les clés de session : une attaque passive est possible sur toutes les communications passées et futures.

Les suites cryptographiques contenant `DHE` ou `ECDHE` reposent sur un échange de clé Diffie-Hellman avec des éléments secrets éphémères, et sur une authentification du serveur explicite avec une signature RSA (ou DSA ou ECDSA) couvrant les paramètres Diffie-Hellman et les aléas échangés. Ces éléments (paramètres DH et signature) sont alors envoyés par le serveur dans le message `ServerKeyExchange`, entre les messages `Certificate` et `ServerHelloDone`, comme le montre la figure 2

L'avantage de Diffie-Hellman éphémère est qu'il permet d'obtenir la propriété de PFS (*Perfect Forward Secrecy*) : la connaissance de la clé privée du serveur n'apporte aucun avantage à un attaquant pour déchiffrer les connexions TLS passées. En revanche, deux attaques sont encore envisageables :

— si le groupe Diffie-Hellman est trop petit (voir section 3.1.4), il est possible de calculer le logarithme discret et de casser le secret partagé ;

— si l'attaquant récupère ou casse la clé privée du serveur, il lui est possible de réaliser une attaque active, de type *man in the middle*, pour usurper l'identité du serveur et manipuler les connexions TLS futures.

`TLS_DHE_RSA_WITH_AES_128_CBC_SHA` est un exemple d'une telle suite cryptographique :

— `DHE` décrit la méthode utilisée pour que les deux parties se mettent d'accord sur la clé de session (on parle en anglais de *Key Agreement*). `DHE` signifie Diffie-Hellman Éphémère ;

— Associée à `DHE`, la partie `RSA` indique que les paramètres Diffie-Hellman du serveur seront signés avec la clé privée RSA associée au certificat envoyé au client. Ce mécanisme permet l'authentification du serveur (en réalité, la signature couvre non seulement ces paramètres, mais également les aléas fournis lors des messages `ClientHello` et `ServerHello`, garantissant le caractère non rejouable de la négociation) ;

— `AES_128_CBC` concerne la protection en confidentialité des données échangées : l'algorithme de chiffrement par bloc AES avec une clé de 128 bits sera utilisé en mode CBC (*Cipher-Block-Chaining*) ;

— `SHA` enfin indique que HMAC SHA1 sera utilisé pour la protection en intégrité.

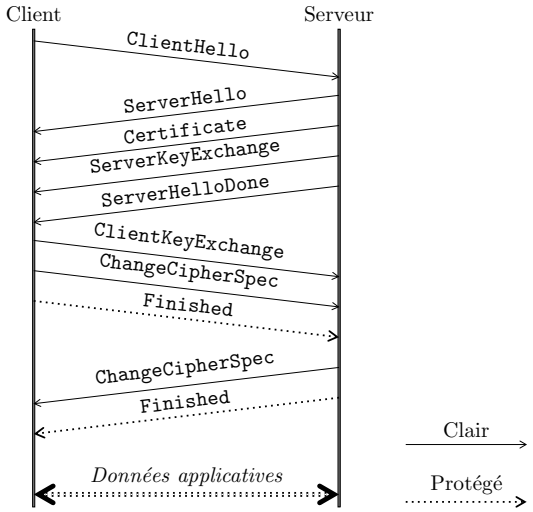

FIGURE 2 – Exemple de négociation TLS avec une suite DHE-RSA.

1.3.2 Modes combinés

Avec TLSv1.2, il est possible d'utiliser un algorithme de chiffrement par bloc dans un mode combiné, c'est-à-dire un mode assurant à la fois la confidentialité et l'intégrité des données traitées. Le mode standardisé est GCM (*Galois Counter Mode*).

1.3.3 Suites cryptographiques « partielles »

Une suite cryptographique peut ne pas offir de protection en confidentialité. Sa dénomination fera apparaître l'algorithme factice `NULL`, comme dans la suite `TLS_RSA_WITH_NULL_SHA`.

De même une suite cryptographique contenant `ADH` correspond à un échange de clé Difie-Hellman anonyme, n'offrant aucune authentification du serveur.

Bien entendu, les clients et serveurs TLS standard n'acceptent pas par défaut de telles suites cryptographiques. Il est cependant utile de tester la configuration d'un serveur avec des outils comme `sslscan` afin de s'assurer que de telles suites ne sont pas proposées (l'annexe C présente rapidement cet outil).

1.3.4 Authentification client

Il est possible pour le serveur d'authentifier le client. Il lui demande alors de présenter un certificat avec le message supplémentaire `CertificateRequest`, et le client envoie deux messages supplémentaires (`Certificate` et `CertificateVerify`) pour présenter son certificat et signer les messages échangés jusque là.

Dans la section 2.3 traitant de la renégociation sécurisée, un exemple est donné où ces messages apparaissent (figure 5 page 10).

1.3.5 Courbes elliptiques

Depuis la RFC 4492 [5], la cryptographie basée sur les courbes elliptiques peut être mise en œuvre par le protocole TLS. Elle peut intervenir lors de l'échange de clé avec l'algorithme ECDHE (il s'agit d'un Diffie-Hellman éphémère réalisé sur une courbe elliptique) ou lors de l'authentification du serveur si ce dernier dispose d'une clé ECDSA.

L'utilisation des courbes elliptique nécessite l'envoi d'extensions TLS pour que le client et le serveur se mettent d'accord sur les courbes supportées. On retrouve ces paramètres sur la page de l'IANA dédiée aux paramètres TLS [17].

Même si l'usage des courbes elliptiques n'est pas extrêmement répandu, les navigateurs web récents proposent d'utiliser des suites contenant ECDHE et ECDSA. Depuis novembre 2011, les services de Google choisissent en priorité les suites cryptographiques `ECDHE_RSA`, qui offrent la propriété de PFS.

2 Choix de la version du protocole

Comme nous l'avons vu, il existe plusieurs versions du protocole, qui sont loin d'avoir les mêmes propriétés de sécurité.

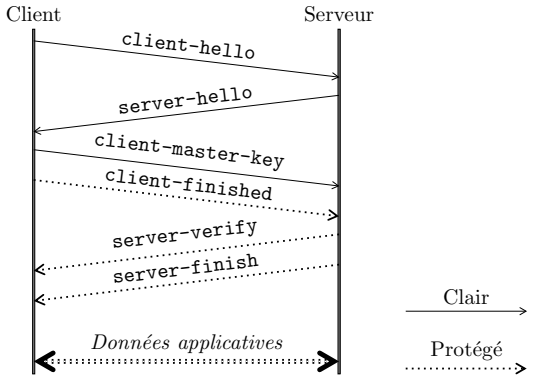

FIGURE 3 – Exemple de négociation SSLv2. Les messages en italiques sont protégés avec les clés négociées.

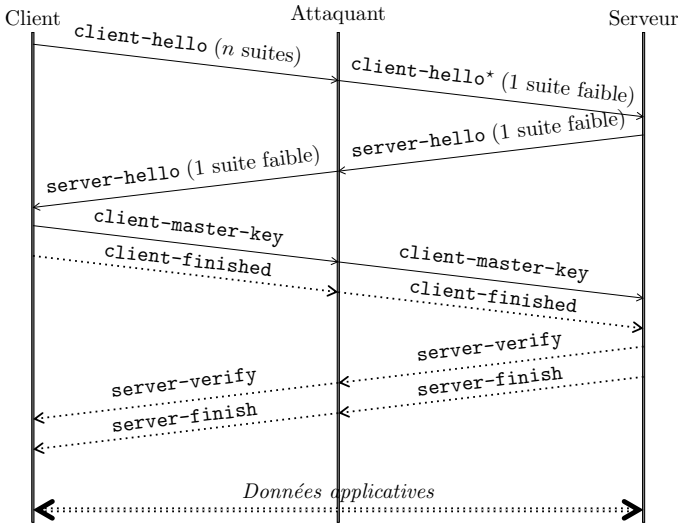

FIGURE 4 – Exemple d'attaque sur la négociation SSLv2. En modifiant le message `client-hello` émis par le client, l'attaquant peut choisir, parmi les suites proposées par le client, la suite cryptographique la plus faible, dès lors que le serveur l'accepte.

En fonctionnement normal, le client envoie un `ClientHello` contenant les numéro de version minimum et maximum qu'il supporte. Si une version parmi celles proposées par le client satisfait le serveur, celui-ci la choisit dans son `ServerHello`.

2.1 Négociation des algorithmes à la baisse avec SSLv2

Les schémas de connexion décrits en introduction (figures 1 et 2) s'appliquent tout aussi bien à SSLv3 qu'aux différentes versions TLS, qui partagent les mêmes concepts et emploient la même terminologie quant aux messages échangés. Il n'en est pas de même de la première version publique du protocole, SSLv2, qui utilise des messages et des mécanismes différents, comme l'indique la figure 3.

Sans évoquer en détails les différences entre d'une part SSLv2 et d'autre part les versions plus récentes du protocole, un point essentiel est à noter : alors que SSLv3 et TLS disposent de messages `Finished` qui protègent en intégrité l'ensemble des messages échangés pendant la négociation, les messages de SSLv2 ne couvrent que les identifiants de session [2] et l'aléa envoyé par le client [3].

Il est donc possible à un attaquant de faire utiliser à un client et à un serveur une suite cryptographique moins forte que celle qu'ils auraient naturellement choisie, comme le montre le schéma 4.

2. L'identifiant de session est un paramètre permettant d'accélérer les connexions futures entre le client et le serveur à l'aide du mécanisme de reprise de session qui n'est pas abordé dans ce document.

3. Le message `server-verify` contient l'aléa du client protégé avec les clés négociées. Les messages `client-finish` et `server-finish` contiennent respectivement l'identifiant de session initial et le nouvel identifiant de session.

Dès qu'un client et un serveur utilisent SSLv2, il leur est impossible de connaître de manière fiable les suites cryptographiques acceptées par leur interlocuteur, puisqu'un attaquant se plaçant entre les deux peut manipuler les messages sans être détecté.

2.1.1 Détection et correction

Un client ou un serveur acceptant une négociation SSLv2 est vulnérable.

En théorie, si le client et le serveur supportent une version postérieure du protocole (SSLv3 ou TLS), il existe un mécanisme permettant au client et au serveur de se rendre compte que SSLv2 a été négocié à tort [4]. Cependant, ce mécanisme est généralement mal implémenté, et ne couvre pas la totalité de la menace.

Pour être exhaustif, SSLv2 présente d'autres problèmes de sécurité :
— les suites cryptographiques SSLv2 utilisent toutes HMAC MD5 pour assurer l'intégrité des messages ;
— SSLv2 utilise la même clé pour protéger le flux en intégrité et en confidentialité ;
— SSLv2 ne propose pas de mécanisme fiable pour signaler la fin d'une connexion, permettant des attaques par « troncature » du flux.

Pour toutes ces raisons, la RFC 6176 [28] interdit aujourd'hui l'utilisation de SSLv2.

La version SSLv2 est dangereuse et ne doit pas être employée.

2.2 Attaque de Bleichenbacher sur PKCS#1 v1.5

En 1998, Daniel Bleichenbacher a publié une attaque cryptographique [8] sur le chiffrement RSA PKCS#1 version 1.5 (décrit dans la RFC 2313 [18]).

Appliquée à TLS, l'attaque repose sur deux hypothèses :
— l'échange de clé s'est fait par chiffrement RSA ;
— le serveur accepte de négocier SSLv2 ou SSLv3.

et comporte deux parties :
— dans un premier temps, l'attaquant capture une connexion SSL/TLS entre un client et un serveur donné ;
— dans un second temps, l'attaquant a accès au serveur qui a été contacté, et monte de nombreuses sessions SSLv2 ou SSLv3 avec lui pour lui envoyer des messages forgés.

À l'aide de ces échanges avec le serveur, l'attaquant va pouvoir déchiffrer le `ClientKeyExchange` envoyé par le client et retrouver le *pre master secret*. Il pourra alors dériver les clés de session et déchiffrer la totalité du trafic capturé.

Plus précisément, l'attaque repose sur la capacité de l'attaquant à distinguer deux cas d'erreur :
— une erreur de bourrage (*padding*) lors du déchiffrement par le serveur du message `ClientKeyExchange` ;
— une erreur ultérieure lors de la vérification du message `Finished`.

Si ces deux cas donnent lieu à des codes d'erreur différents, l'attaquant peut envoyer des requêtes auprès du serveur contenant des versions altérées du `ClientKeyExchange` initial, et obtenir des informations sur le contenu clair du message.

En réalité, il est nécessaire que les deux comportements (erreur de *padding* et erreur de vérification) soient complètement indiscernables du point de vue de l'attaquant. Pour garantir cela, il faut non seulement répondre le même code d'erreur dans tous les cas, mais aussi prendre en compte des canaux cachés comme le temps de réponse.

Le nombre de requêtes nécessaires pour monter cette attaque dépend de la taille de la clé privée RSA du serveur. Il en faut de l'ordre d'un million pour une clé RSA de 1024 bits, d'où son nom de *Million Message Attack*.

2.2.1 Détection et correction

Comme l'attaque repose sur la capacité de l'attaquant à distinguer deux types d'erreur dans le déchiffrement, il faut rendre le comportement du serveur indépendant du type d'erreur. Pour TLS, la RFC définissant TLSv1.0 propose une implémentation qui corrige effectivement le fonctionnement du serveur pour que les deux comportements soient indiscernables.

Du point de vue du serveur, il est possible de détecter une attaque en cours en surveillant le nombre de négociations TLS avortées.

Si un client souhaite détecter si un serveur est vulnérable, il peut monter des sessions TLS envoyer des messages incorrects destinés à déclencher les deux cas d'erreur (erreur sur le *padding* et erreur sur le déchiffrement

4. Comme SSLv2 ne supporte que l'échange de clé par chiffrement RSA, le mécanisme consiste à intégrer dans le *pre master secret* la version maximale supportée par le client. Comme l'attaquant ne peut modifier le message sans faire échouer la connexion, le serveur reçoit nécessairement l'information. S'il connaît SSLv3 au moins, il peut alors vérifier si le client avait proposé mieux que SSLv2.

du message `Finished`) pour voir si les deux situations sont distinguables. Cette méthode est cependant intrusive et s'apparente à l'attaque elle-même.

Il semble donc raisonnable d'imposer que le protocole utilisé soit au moins TLSv1.0 dont l'implémentation impose une contre-mesure. Aucune négociation ne devrait aboutir à une version antérieure.

> Seules des implémentations conformes à l'implémentation décrite dans TLSv1 doivent être employées. Les suites offrant la PFS (non affectées) doivent être favorisées.

2.3 Renégociation sécurisée

Nous avons vu plus haut comment se déroulait la négociation entre le client et le serveur pour l'établissement de clés de session. La norme TLS propose également un mécanisme pour renégocier les algorithmes et les paramètres cryptographiques. Des raisons légitimes pour une telle renégociation sont :

— le raffraîchissement des clés cryptographiques avant leur usure ;
— après une première négociation sans authentification du client, le serveur initie une renégociation pour demander un certificat au client afin d'authentifier l'accès à du contenu protégé ;
— la nécessité d'employer des algorithmes différents en fonction des données échangées (ce dernier cas est généralement une mauvaise raison).

La renégociation peut se faire à l'initiative du client (qui envoie directement un nouveau `ClientHello`) ou du serveur (qui sollicite le client par un message `HelloRequest`), et les messages échangés sont identiques à ceux d'une négociation initiale, à la différence près qu'elle profite du canal chiffré intègre mis en place par la négociation précédente.

Par exemple, le schéma 5 montre les échanges réseau correspondant à une renégociation initiée par le serveur imposant a posteriori une authentification client. Les nouveaux messages sont en gras :

— pour provoquer la renégociation, le serveur envoie un `HelloRequest` ;
— dans le cas présenté, le serveur utilise cette nouvelle négociation pour demander un certificat client (message `CertificateRequest`) auquel le client répond avec deux messages supplémentaires (`Certificate` et `CertificateVerify`).

2.3.1 Description de l'attaque de Ray et Dispensa

Le 4 novembre 2009, deux chercheurs, Marsh Ray et Steve Dispensa, ont publié des éléments démontrant l'existence d'une faille conceptuelle dans la gestion de la renégociation des sessions TLS [24]. Plusieurs scénarios ont été décrits, mettant en jeu un client et un serveur légitimes, ainsi qu'un attaquant réseau actif.

L'attaquant va se placer au milieu et utiliser le fait que les différentes négociations réalisées sur une même connexion TCP ne sont pas liées entre elles. La figure 6 permet de suivre les messages échangés. Elle met en œuvre deux négociations :

— la première est initiée par le client et notée `(C)` ;
— la seconde est initiée par l'attaquant et notée `(A)`.

Lorsque le client légitime essaie d'initier une session TLS avec le serveur TLS (message `ClientHello (C)`), l'attaquant bloque le paquet correspondant et le conserve. Il entame alors une connexion TLS avec le serveur, correspondant aux messages `(A)`, qui représentent la session initiale vue par le serveur. L'attaquant peut alors injecter des messages clairs à destination du serveur (message *Charge*).

Ensuite, l'attaquant émet le paquet initial `ClientHello (C)` du client à travers le canal sécurisé par la session `(A)` pour initier la renégociation. Le client légitime et le serveur vont alors négocier de nouveaux paramètres, sauf qu'il s'agira de la session initiale pour le client (les messages `(C)` passent en clair entre le client et l'attaquant) et d'une renégociation pour le serveur (ces mêmes messages sont chiffrés par l'attaquant avec les paramètres de la session `(A)`).

Une fois les nouveaux paramètres négociés, l'attaquant n'a plus accès aux données, mais continue de router les paquets chiffrés entre le client et le serveur.

2.3.2 Impact de l'attaque

Elle permet à un attaquant d'avoir un dialogue avec le serveur avant de laisser le client prendre la suite. Dans l'exemple donné sur la figure, le serveur croit recevoir la concaténation de la charge et des paquets du client sans qu'aucun ne puisse le détecter. L'impact de cette attaque dépend donc fortement de la nature des données applicatives émises à travers le canal chiffré.

Parmi les analyses réalsiées sur le sujet, l'analyse de Thierry Zoller [29] est la plus complète. La majorité des attaques envisagées portent sur HTTPS. D'autres protocoles comme SMTPS et FTPS pourraient être affectés dans certaines situations, mais les implémentations étudiées ne semblent pas vulnérables. Quelques exemples concernant HTTPS sont décrits ci-dessous.

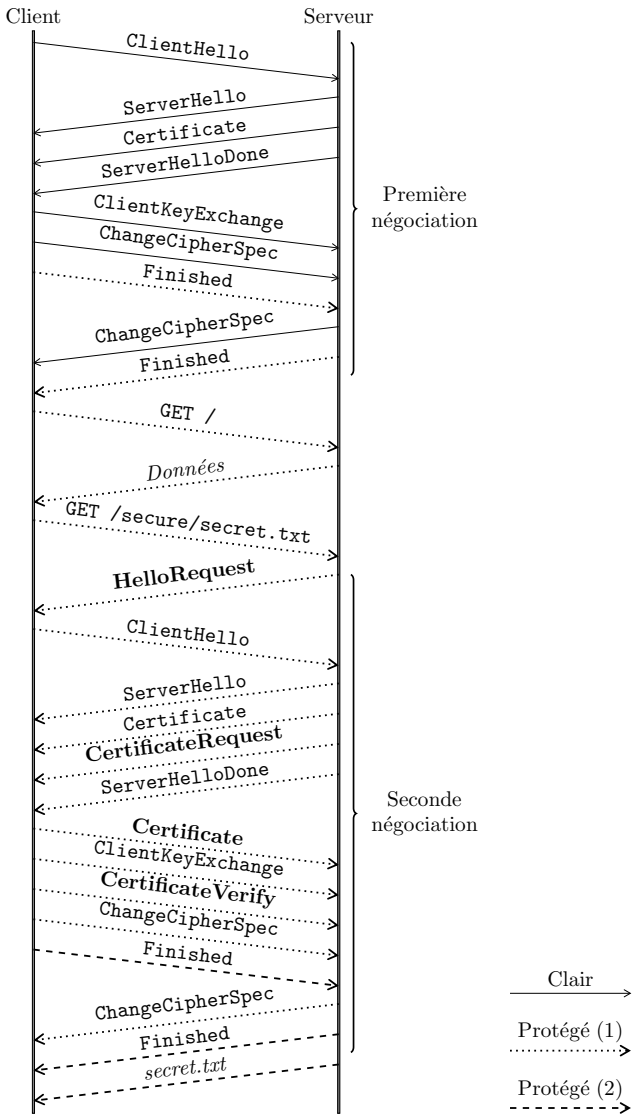

FIGURE 5 – Exemple d'utilisation de la renégociation dans le cas de HTTPS. Après une première négociation simple, le client demande l'accès à une ressource protégée, ce qui déclenche une demande de renégociation de la part du serveur qui exige cette fois la présentation d'un certificat client avant d'émettre sa réponse.

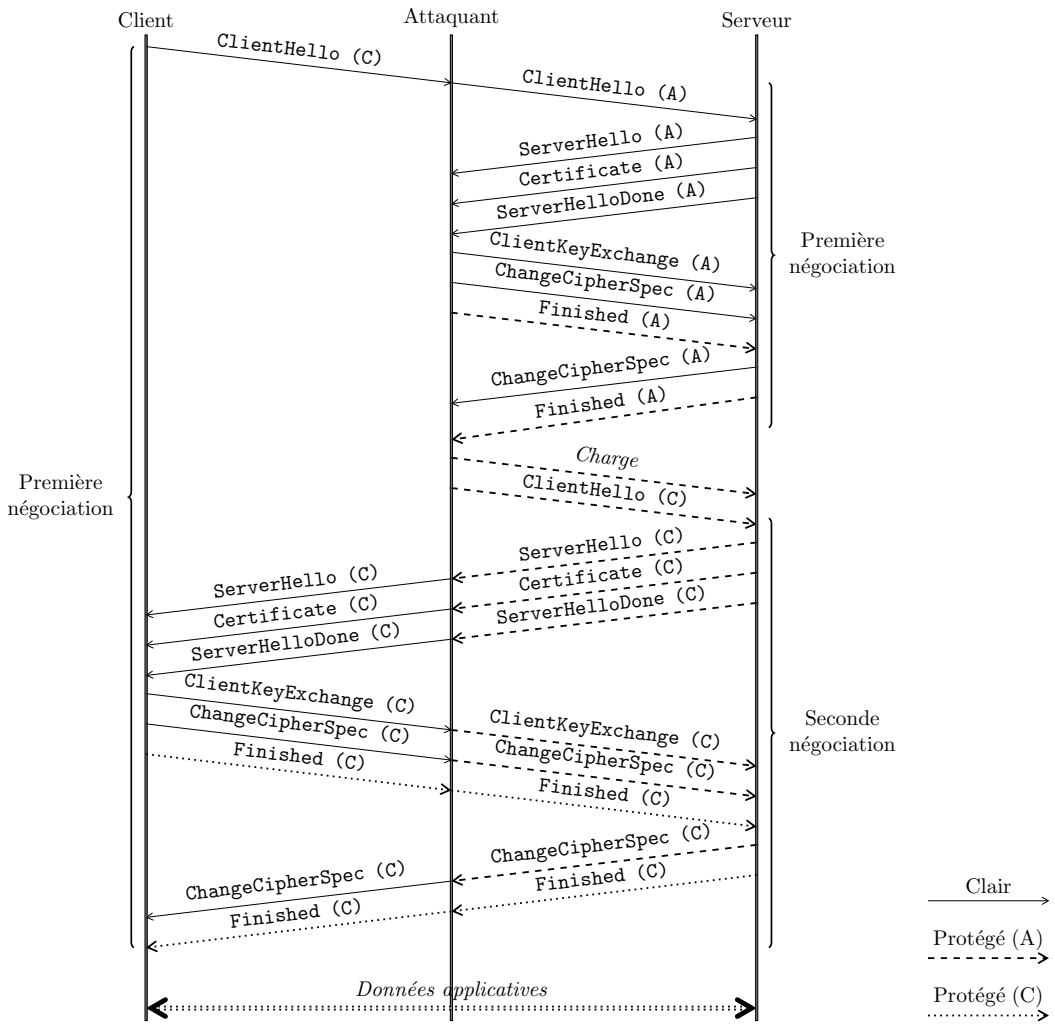

FIGURE 6 – Détails de l'attaque de Ray et Dispensa sur la renégociation.

Attaques proches d'une CSRF L'attaquant peut profiter de cette attaque pour injecter un début de requête nécessitant une authentification, puis laisser le client ajouter les en-têtes nécessaires à l'authentification. De cette manière, il pourra exécuter du code côté serveur avec les droits du client.

Par exemple, l'attaquant peut envoyer le contenu suivant afin de forcer le changement de mot de passe sur un site donné :

```
GET /change-mot-de-passe.php?mdp=secret
X-Forget-This:
```

sans mettre de retour charriot à la fin de la deuxième ligne. Ensuite, le client envoie légitimement le contenu suivant, avec un *cookie* l'authentifiant :

```
GET /index.php
Cookies: login=georges;session=03462f4a47cd67f
```

Le serveur recevra donc le contenu ci-dessous, sans pouvoir distinguer ce qui vient du client et ce qui vient de l'attaquant :

```
GET /change-mot-de-passe.php?mdp=secret
X-Forget-This: GET /index.php
Cookies: login=georges;session=03462f4a47cd67f
```

L'attaque aura les mêmes effets qu'une attaque de type *Cross-Site Request Forgery* (CSRF), en offrant à l'attaquant plus d'expressivité, puisqu'il n'est pas limité à la commande `GET`. Les applications vulnérables à cet exemple peuvent donc être corrigées à l'aide de jetons anti-CSRF.

Mettre en œuvre l'attaque revient à forcer un utilisateur à s'authentifier sur un site vulnérable, puis à détourner une connexion TLS dès son démarrage une fois la victime authentifiée. Un peu d'ingénierie sociale permet aisément de remplir cette hypothèse.

Utilisation de la méthode TRACE Lorsque la méthode TRACE est acceptée par un serveur, l'attaquant peut émettre des données qui seront réémises par le serveur. Il est donc possible d'émettre auprès du serveur la requête suivante (la partie injectée par l'attaquant est en italique) :

```
TRACE / HTTP/1.0
<HTML><script>alert('echo')</SCRIPT></HTML>:
X-IGNORE: GET /index.php
Host: www.service.com
```

qui produira la réponse suivante de la part du serveur :

```
HTTP/1.0 200 OK
[...]
Content-Type: message/http

TRACE / HTTP/1.0
<HTML><script>alert('echo')</SCRIPT></HTML>:
X-IGNORE: GET /index.php
Host: www.service.com
```

Malgré l'en-tête `Content-type`, certains navigateurs interprèteront cette réponse comme une page HTML légitime, et donc mener à l'exécution de JavaScript associé à l'origine `www.service.com` côté client. Heureusement, la méthode `TRACE` est traditionnellement désactivée.

Vol d'authentifiants Enfin, une autre application est le vol d'authentifiants. L'attaquant peut insérer une requête `POST` consistant en l'envoi d'un message que la victime viendra remplir avec ses authentifiants. Cette fois, l'ensemble des données fournies par la victime sont vues par le serveur comme le contenu du message.

```
POST send.php
Authorization: Basic <authentifiants de l'attaquant>

<Requête initiale de la victime>
Authorization: Basic <authentifiants de la victime>
User-Agent: <navigateur de la victime>
[...]
```

Dans le cas de Twitter, il était ainsi possible d'obtenir les 140 premiers octets de la requête du client (la partie émise par l'attaquant est en italique, le reste provient du client légitime), en particulier la chaîne de caractères `Authorization: Basic [...]` contenant le mot de passe de la victime, qui étaient alors publiés sur le compte Twitter de l'attaquant. Cette vulnérabilité particulière a été corrigée par Twitter dès sa publication.

2.3.3 Détection et correction

Face à cette menace, une première contre-mesure était de désactiver la renégociation côté serveur. En revanche, interdire la renégociation dans les clients était inutile.

```
Secure Sockets Layer
    SSL Record Layer: Handshake Protocol: Client Hello
        Content Type: Handshake (22)
        Version: SSL 3.0 (0x0300)          # Version minimum
        Length: 83
        Handshake Protocol: Client Hello
            Handshake Type: Client Hello (1)
            Length: 79
            Version: TLS 1.0 (0x0301)      # Version maximum
```

FIGURE 7 – Détails d'un en-tête TLS.

Afin de réellement corriger le problème, une nouvelle extension TLS a été définie dans la RFC 5746 [25], permettant de signifier au cours de la négociation quelles sont les négociations passées que chaque partie a vues de son côté. En cas de désaccord, le client et le serveur légitimes doivent mettre fin à la communication.

Cette correction utilise le mécanisme des extensions TLS, qui n'est compatible qu'avec TLSv1.0 et les versions suivantes, et qui peut casser la compatibilité avec certaines implémentations exotiques. Un moyen a donc été imaginé pour que le client puisse signaler sans risque qu'il supporte cette nouvelle extension, la renégociation sécurisée, en proposant dans sa liste de suites cryptographiques une suite factice (`TLS_EMPTY_RENEGOTIATION_INFO_SCSV`).

Cependant, seules des implémentations à jour compatibles avec TLSv1.0 peuvent s'entendre pour activer cette extension et garantir l'absence de l'attaque présentée plus haut.

Concernant la détection, il est simple de savoir si un client ou un serveur sont vulnérables : dès qu'ils acceptent de monter une connexion SSL/TLS autorisant ne mettant pas en œuvre la nouvelle extension, les attaques sont envisageables. Au-delà, il n'y a pas de moyen générique de vérifier que la vulnérabilité est exploitée.

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

2.4 Problèmes liés aux mécanismes de *fallback*

En théorie, l'enveloppe externe, appelée *record*, du `ClientHello` peut indiquer une version de SSL/TLS plus vieille que le contenu du message. Une interprétation communément admise est que le client émettant un tel `ClientHello` supporte l'ensemble des versions comprises entre les deux numéros. Dans l'exemple de la figure 7, le client émet un paquet SSLv3 dans lequel il indique être compatible avec TLSv1.0.

Cependant, certains serveurs ont des comportements incohérents avec la norme lorsque le `ClientHello` qu'ils reçoivent contient des extensions TLS ou une version trop récente du protocole.

Pour pouvoir tout de même communiquer avec de tels serveurs, les navigateurs recourent à un mécanisme de *fallback* :

— le client émet un premier `ClientHello` faisant état de la version maximale qu'il supporte ;
— en cas de rejet du message, le navigateur essaie ensuite de renvoyer un `ClientHello` avec une version inférieure (et moins d'extensions) ;
— après avoir épuisé toutes les versions supportées, si le message est toujours rejeté, la connexion est refusée.

Par exemple, Firefox utilise ce mécanisme en essayant dans l'ordre les `ClientHello` suivants :

— message TLSv1.0 avec des extensions (courbes elliptiques, ticket de session, *Server Name Indication*) et de nombreuses suites cryptographiques ;
— message SSLv3 sans extensions et avec un sous-ensemble des suites cryptographiques précédentes.

Safari sur iPhone ajoute un autre `ClientHello` avec une version TLSv1.2 et encore plus de suites cryptographiques.

Il est important de noter que le rejet du serveur consiste simplement en une alerte émise en clair, qu'un attaquant peut facilement contrefaire. Ainsi, il est possible de réaliser une négociation à la baisse indétectable sur la version du protocole négociée. Les impacts en sont les suivants :

— l'attaquant peut forcer le client et le serveur à ne pas utiliser les extensions TLS, ce qui permet d'interdire l'utilisation de courbes elliptiques ;
— l'attaquant peut restreindre les suites cryptographiques à un sous-ensemble de compatibilité.

Afin de contrer cette attaque qui supprime en pratique toute extensibilité au protocole SSL/TLS, des discussions ont eu lieu sur la liste du groupe de travail IETF sur TLS. La bonne méthode pour éviter les problèmes serait d'envoyer un unique `ClientHello` contenant directement les paramètres souhaités par le client (version de TLS maximum supportée, extensions), et de contacter les administrateurs des serveurs récalcitrants. Cependant, comme l'indique une étude sur les pratiques HTTPS [1], nombreux sont les serveurs incompatibles. Pour être

acceptée, cette méthode nécessiterait donc en pratique le maintien, au moins de manière temporaire, d'une base de données de ces serveurs.

2.5 Recommandations

Il existe différentes failles relatives à la version du protocole SSL/TLS utilisée, et pour éviter d'être affectés, des logiciels doivent suivre les recommandations suivantes :

— afin de rendre inopérante l'attaque de Bleichenbacher et de l'afficher clairement au client, **un serveur ne doit accepter que des versions supérieures ou égales à TLSv1.0 qui implémentent les contre-mesures** ;
— de même, pour bénéficier de la renégociation sécurisée, il est nécessaire d'avoir le support des extensions, qui nécessite TLSv1.0 au moins ;
— la renégociation sécurisée doit être supportée et imposée côté client et côté serveur ;
— étant donné ce qui précède, **un client ne devrait jamais accepter une session dans laquelle la version négociée est inférieure à TLSv1.0** ;
— certains clients émettent encore des `ClientHello` de compatibilité SSLv3, ce qui réduit l'extensibilité du protocole (extensions, suites cryptographiques). Si les recommandations précédentes sont appliquées, ce comportement est inutile (puisque le client refusera une version antérieure à TLSv1.0 et supportera la renégociation sécurisée), mais peut être toléré.

2.6 Test des implémentations

L'annexe A page 25 indique comment tester les versions supportées par un client ou un serveur à partir d'une version recompilée d'OpenSSL.

Globalement, les piles TLS reconnaissent SSLv3 et TLSv1.0. Par défaut, aucune pile testée n'accepte SSLv2. De plus, Safari supporte les versions récentes de TLS (1.1 et 1.2). Internet Explorer sous Windows 7 et Opera peuvent également être configuré pour accepter les versions 1.1 et 1.2.

Côté serveur, jusqu'à récemment, Apache2 muni de `mod_ssl` supportait par défaut SSLv3 et TLSv1.0. Les versions récentes (2.2.22 sous Debian) sont liées à une version d'OpenSSL récente (1.0.1) et acceptent TLSv1.2 lorsque c'est proposé par le client. Apache2 avec `mod_gnutls` supporte depuis quelques années SSLv3 à TLSv1.2.

Enfin, la renégociation sécurisée est supportée par l'ensemble des clients et serveurs testés.

En conclusion, il est possible de respecter les recommandations ci-dessus en désactivant SSLv3 dans les logiciels utilisés et en utilisant des versions à jour puisque la renégociation sécurisée est désormais supportée

3 Choix des suites cryptographiques

Lors de la négociation de la suite cryptographique, quatre algorithmes sont négociés :

— l'algorithme d'authentification des parties ;
— le mécanisme d'échange de clé ;
— la protection des données en confidentialité ;
— la protection des données en intégrité.

En passant en revue ces différents points, nous allons déterminer les suites cryptographiques à éviter.

3.1 Mécanismes d'authentification et d'échange de clés

Classiquement, ces deux fonctions sont regroupées, car elles reposent toutes les deux sur de la cryptographie asymétrique, et parfois sur un même algorithme.

3.1.1 Premier tri dans les suites

Comme indiqué dans la section 1.3, il existe certaines suites cryptographiques n'offrant aucune authentification (suites contenant `ADH`, pour *Anonymous Diffie Hellman*). Si un client accepte de négocier une telle suite, un attaquant capable de monter un *man-in-the-middle* au niveau du réseau peut se faire passer pour le serveur de manière triviale, et lire et modifier l'intégralité du flux TLS à destination du serveur légitime.

Dans cet article, nous ne parlerons pas de certains mécanismes :

— Kerberos (suites `TLS_KRB5_*`) et les mécanismes reposant sur un mot de passe ou un secret pré-partagé (`TLS_PSK_*` et `TLS_SRP_*`), qui sont très peu présents dans l'écosystème TLS ;
— les suites utilisant un échange de clé Diffie-Hellman fixe (`TLS_DH_*` et `TLS_ECDH_*`) : la partie privée du serveur est constante et la partie publique est incluse dans le certificat présenté. Là encore, il s'agit de suites peu présentes dans la nature, auxquelles on préfèrera les versions éphémères (`DHE` et `ECDHE`) ;

Une fois toutes ces suites écartées, il reste deux familles de suites cryptographiques :

— `TLS_RSA_*`, les suites reposant sur le chiffrement RSA tel que présenté à la figure 1, qui n'offre pas de PFS (*Perfect Forward Secrecy*) ;
— `TLS_(EC)DHE_RSA_*`, `TLS_DHE_DSS_*` et `TLS_ECDHE_ECDSA_*`, les suites reposant sur un échange Diffie-Hellman éphémère (éventuellement sur courbe elliptique) signé par RSA, DSA ou ECDSA, qui offrent la PFS (figure 2).

3.1.2 Discussion sur la taille des clés

Parmi les suites restantes, les algorithmes utilisés reposent sur des problèmes mathématiques difficiles : le logarithme discret ou la factorisation des entiers. Le temps nécessaire à retrouver une clé privée à partir de la clé publique dépend du domaine sur lequel les calculs sont réalisés :
— $\mathbb{Z}/n\mathbb{Z}$, les entiers modulo, pour RSA, DH et DSA ;
— les courbes elliptiques pour ECDSA, ECDH.

Dans le premier cas, les records concernant des clés RSA cassées sont autour de 768 bits [19], et 1024 bits semble accessible. Pour Diffie-Hellman, le record est de 530 bits sur un corps premier. C'est pourquoi pour les algorithmes utilisant les entiers modulo, il est recommandé d'utiliser des clés de 2048 bits, voire 4096 bits pour un usage à long terme (au-delà de 2020).

Pour les courbes elliptiques, la taille des clés est significativement plus petite, puisque le record du logarithme discret concerne une courbe sur 112 bits. Les recommandations indiquent qu'une taille de clé de 200 bits est suffisante pour un usage à court terme. On exigera 256 bits pour des usages à plus long terme (au-delà de 2020).

L'ensemble des recommandations ci-dessus sont issues de l'annexe B.1 du Référentiel Général de Sécurité [2].

Concernant DSA, il est intéressant de remarquer que de nombreuses implémentations suivent encore aujourd'hui la norme initiale décrivant l'algorithme [22] et non la dernière révision [23]. Cela pose deux problèmes conduisant à écarter les suites utilisant DSA :
— la taille des clés est limitée à 1024 bits ;
— la fonction de hachage imposée dans la signature du certificat est SHA1, ce qui est fortement déconseillé aujourd'hui (voir section 4.1).

3.1.3 Divulgation de la clé privée serveur

Il y a plusieurs cas menant à la divulgation de la clé privée associée au certificat du serveur.

Utilisation d'une clé de petite taille Une première possibilité est qu'un attaquant casse la clé privée à partir de la clé publique. Comme indiqué ci-dessus, cela n'est bien entendu possible que pour des clés dont la taille est trop petite. C'est le cas d'une clé privée RSA 512 bits, comme le montrent les exemples suivants.

En juillet 2009, Benjamin Moody a publié la clé privée de signature du code utilisée pour authentifier les mises à jour des calculatrices Texas Instrument (série TI83+). Il lui avait fallu 73 jours de calculs. Les clés correspondant aux autres calculatrices ont ensuite été rapidement cassées par *RSA Lattice Siever*, un projet de calcul distribué [26].

En novembre 2011, une AC intermédiaire malaise (Digicert Sdn. Bhd.) a révoqué 22 certificats contenant des clés RSA de 512 bits. Ces clés auraient en effet permis à un attaquant d'usurper les sites légitimes (dont des sites du gouvernement malais). De plus, ces certificats ne possédaient pas d'extension sur leur usage ou leur révocation, rendant en particulier leur révocation inefficace. Pour résoudre le problème, Entrust, l'autorité racine ayant signé l'autorité malaise, a révoqué cette dernière. Les certificats révoqués ont également été fournis aux éditeurs de navigateurs pour qu'ils soient éventuellement intégrés dans des listes noires [14] (le mécanisme de listes noires pour les certificats n'est pas standard, mais il est aujourd'hui nécessaire dans les navigateurs pour pallier l'inefficacité de la révocation en pratique, voir section 4.5).

Mauvaise gestion de l'accès aux secrets Un serveur en fonctionnement a besoin d'avoir accès au service de déchiffrement pour mettre en œuvre SSL. Cela passe souvent par la présence en clair de la clé privée sur le serveur[5]. Dans ce cas, il est nécessaire d'assurer que l'accès à ce fichier soit limité au serveur SSL.

Cela passe par une isolation des services fournis sur la machine pour éviter qu'une faille sur l'application permette la récupération de la clé privée. On pourra utiliser la gestion des droits offerts par le système d'exploitation utilisé et des mécanismes de cloisonnements entre la partie SSL et la partie applicative.

De plus, il est important d'assurer la sécurité physique à la machine : en général, avoir un accès physique au serveur permet trivialement de récupérer les clés privées stockées.

Bien entendu, on évitera soigneusement de mettre la clé privée sur des sites de partage à accès public tel que `pastebin.com`.

5. L'utilisation d'un HSM (*Hardware Security Module* pour stocker la clé privée permet de déplacer le problème sur des équipements dont le rôle est de servir de coffre-forts de clés.

Défaut dans le générateur d'aléa Si le générateur d'aléa utilisé pour créer la clé cryptographique est prédictible, il est possible à un attaquant de regénérer la même clé en reproduisant les conditions de la génération initiale.

Entre septembre 2006 et mai 2009, la version Debian d'OpenSSL contenait une erreur ramenant l'entropie du générateur d'aléa à quelques dizaines de bits. L'ensemble des clés possibles pouvaient alors aisément être énumérées en quelques heures, rendant toutes ces clés disponibles à un attaquant.

Plus récemment, suite à la publication de données volumineuses par l'Electronic Frontier Foundation (EFF), des chercheurs ont constaté qu'il était possible de factoriser certains modules RSA simplement en calculant leur PGCD [3]. Lenstra et al. ont ainsi montré qu'en présence d'un mauvais générateur d'aléa, certains équipements pouvaient dans leur vie générer d'une part le module RSA $N_1 = pq$ et d'autre part le module $N_2 = pr$. Puisque ces deux modules partagent un facteur premier commun, celui-ci peut être découvert en appliquant l'algorithme d'Euclide, menant à la factorisation de N_1 et N_2.

Impact Comme indiqué dans la section 1.3, l'impact de la compromission de la clé privée dépend des suites cryptographiques utilisées :

— en l'absence de PFS (avec les suites `TLS_RSA_*`), toutes les connexions TLS protégées par cette clé privée peuvent être déchiffrées, y compris celles qui auraient pu être capturées avant la compromission de la clé ;
— avec les suites offrant la PFS, une attaque active est nécessaire de la part de l'attaquant, qui ne peut donc déchiffrer que les connexions postérieures à la divulgation de la clé.

Dans tous les cas, la possession de la clé privée permet de monter une attaque active pour lire et modifier le contenu de la communication.

Il est donc nécessaire d'utiliser des tailles de clés suffisantes, de veiller à la protection en fonctionnement des clés privées, et de s'assurer que les clés sont générées à l'aide d'un générateur d'aléa de bonne qualité. L'annexe B1 du RGS [2] contient des recommandations sur l'architecture des générateurs d'aléa.

La section 4.3 traite de l'impact de la divulgation d'une clé privée associée à une autorité de certification.

3.1.4 Utilisation d'un groupe DH trop petit

Considérons maintenant qu'un client et un serveur ont négocié une suite cryptographique acceptable telle que `TLS_DHE_RSA_WITH_AES_128_CBC_SHA`, qui contient des algorithmes acceptables, mais que l'échange Diffie-Hellman est réalisé en utilisant un groupe d'une taille insuffisante. Un attaquant pourrait alors capturer les communications utilisant un tel groupe, calculer le logarithme discret hors ligne, et en déduire le secret partagé. La confidentialité de l'échange n'est plus garantie.

Des groupes Diffie-Hellman ayant une taille de 256 bits ont été détectés dans les données de l'EFF. Le problème a été abordé dans un certain nombre de tickets d'incidents Mozilla et ont fait l'objet du CVE-2010-3173 [21]. Pour réduire le risque d'attaque utilisant ce vecteur, de nombreuses implémentations imposent une taille de groupe Diffie-Hellman minimale (par exemple, Firefox impose une taille d'au moins 512 bits, ce qui est encore en-deçà des recommandations actuelles qui prônent des groupes de 2048 bits).

En revanche, si un attaquant souhaite modifier le contenu du trafic, il lui faudra être capable de casser le logarithme discret en temps réel, ce qui n'est réaliste qu'avec une taille de groupe minuscule.

Concernant l'authentification et l'établissement des clés, il faut :
- utiliser des suites cryptographiques dont l'authentification est assurée par RSA ou ECDSA ;
- générer des biclés correspondant à une taille suffisante ;
- s'assurer que le biclé est généré à l'aide d'un matériel ou d'un logiciel doté d'un générateur aléatoire de bonne qualité ;
- protéger durant toute sa vie la clé privée de manière à éviter sa divulgation ;
- préférer, lorsque cela est possible, des suites cryptographiques assurant la PFS (Diffie-Hellman éphémère) ;
- s'assurer le cas échéant de la taille suffisante des groupes utilisés.

3.2 Algorithme de chiffrement

3.2.1 Premier tri sur les algorithmes disponibles

Certaines suites offrent peu ou pas de protection en confidentialité. Les suites cryptographiques suivantes sont donc à rejeter :

— les suites ne proposant pas de chiffrement (chiffrement `NULL` dans le nom de la suite) ;

— les suites marquées `EXPORT`, qui datent de l'époque où la législation américaine interdisait l'export de logiciels cryptographiques utilisant des clés de chiffrement de plus de 40 bits [6] ;
— les suites proposant des algorithmes de chiffrement obsolètes (RC2, DES, IDEA).

Parmi les suites restantes, on trouve des algorithmes de chiffrement par bloc (AES, Camellia, 3DES) et un algorithme de chiffrement par flot (RC4).

3.2.2 Particularité du mode CBC : attaque à clair choisi sur l'utilisation d'un IV implicite

En 2002, Philippe Rogaway a décrit une attaque concernant l'utilisation d'un IV implicite dans le mode CBC [9].

On suppose qu'un attaquant peut observer les messages chiffrés $C_0 \dots C_k$. Il souhaite déchiffrer le bloc P_i correspondant au chiffré C_i du message chiffré en mode CBC. De plus, il peut choisir une partie des messages clairs à chiffrer de manière adaptative. Comme le chaînage utilise le dernier bloc de chiffré comme IV suivant, le prochain bloc clair P_{k+1} sera d'abord combiné (avec un XOR) avec le chiffré précédent (C_k) puis chiffré avec la primitive de chiffrement par bloc.

Il est alors possible à l'attaquant de tester si P_i vaut une valeur donnée x de la manière suivante : il choisit le bloc clair égal à $x \oplus C_k \oplus C_{i-1}$. Le chiffré vaudra alors $C_{k+1} = E(x \oplus C_{i-1})$. Or, par définition du mode CBC, $C_i = E(P_i \oplus C_{i-1})$. Ainsi, si les chiffrés C_{k+1} et C_i coïncident, cela signifie que P_i vaut bien la valeur x devinée par l'attaquant.

Cette attaque peut sembler totalement irréaliste car elle nécessite que l'attaquant teste a priori l'ensemble des valeurs de x possibles pour trouver de l'information, et qu'il s'agit d'une attaque à clair choisi adaptatif (puisque l'attaquant doit observer le flux chiffré en parallèle). L'attaque a pourtant été corrigée en 2006 dans TLSv1.1 [11] en imposant l'utilisation à chaque message d'un IV explicite. La vulnérabilité étant considérée comme théorique, elle a tardé à être corrigée dans les faits.

La situation a changé en 2011 quand Thai Duong et Juliano Rizzo ont montré à l'aide d'une attaque intitulée BEAST que ces hypothèses pouvaient en réalité être réunies dans un navigateur web [**?**]. En effet, ils ont constaté que les connexions multiples vers un même site HTTPS pouvaient être multiplexées dans une même connexion TLS, indépendamment de l'onglet d'origine du message, rendant possible l'injection de clair choisi dans le canal chiffré. Pour rendre l'attaque possible, il leur a fallu trouver une méthode pour outrepasser le mécanisme de *Same Origin Policy* [7] et constater que dans le cas HTTPS, une partie du bloc à deviner pouvait être connue de l'attaquant en injectant des *cookies* connus. Le problème se ramenait à deviner non plus un bloc complet comme ci-dessus mais uniquement un octet à la fois.

Bien que complexe à mettre en œuvre, cette attaque a fait couler beaucoup d'encre puisque les chercheurs montraient dans leur démonstration comment récupérer un *cookie* d'authentification PayPal. Pour éviter d'utiliser le mode CBC, certains chercheurs ont proposé d'utiliser RC4 en priorité, ce qui n'est pas très satisfaisant pour des raisons évoquées plus bas. Par ailleurs, plusieurs développeurs ont proposé une méthode pour réduire l'impact de l'attaque [8] sans casser la compatibilité avec les implémentations existantes. Enfin, TLSv1.1 commence à faire son apparition dans OpenSSL, ce qui permettra bientôt une véritable correction de la vulnérabilité.

En conclusion, ces vulnérabilités sur le mode CBC sont inexploitables en pratique avec des implémentations à jour des piles SSL/TLS.

Les suites acceptables du point de vue de la confidentialité des données applicatives sont :
— celles utilisant un algorithme de chiffrement par bloc récent, avec une taille de bloc et une taille de clé suffisamment grandes (AES et Camellia). Généralement, le mode opératoire utilisé est CBC, mais TLSv1.2 permet l'utilisation de modes combinés (chiffrement et intégrité).
— celles reposant sur des algorithmes pour lesquels des attaques statistiques existent (RC4, 3DES). L'utilisation de ces suites n'est pas recommandée, mais peut être tolérée pour des raisons de compatibilité si la quantité de données chiffrées entre deux rafraîchissements des clés n'excède pas une certaine taille (quelques centaines de Mo).

3.3 Algorithme d'intégrité

Il n'existe pas d'équivalent à l'algorithme de chiffrement `NULL` : toutes les suites cryptographique offrent un algorithme d'intégrité [9].

On peut classer en plusieurs catégories les algorithmes existants :

6. Cette limite de 40 bits s'appliquait aux algorithmes de chiffrement symétrique. Une restriction équivalente s'appliquait aux algorithmes de chiffrement asymétrique, avec une taille de 1024 bits.

7. Les méthodes employées à cette fin reposait sur l'utilisation d'une applet Java ou sur une version obsolète des *WebSockets*.

8. La méthode consiste, pour envoyer un message $M = m_0 m_1 \dots m_n$, à envoyer d'abord le premier octet m_0 puis à chiffrer le reste. De cette manière, la majorité du message $m_1 \dots m_n$ sera chiffrée avec un IV randomisé.

9. Il existe une exception : la suite factice `TLS_NULL_WITH_NULL_NULL` qui sert à la première négociation en clair et ne doit jamais être négociée.

— les algorithmes de chiffrement par blocs en mode combiné, disponible uniquement avec TLSv1.2 ;
— HMAC SHA256 et HMAC SHA384, également apparu avec TLSv1.2 ;
— HMAC SHA1, qui est acceptable malgré les attaques en collision publiées depuis 2005. En effet, trouver une collision sur SHA1 n'a pas d'impact sur la sécurité de HMAC ;
— HMAC MD5, qu'il est recommandé d'abandonner puisqu'il existe des attaques en collision et en pré-image sur cette fonction.

Pour assurer l'intégrité des données, on utilisera de HMAC SHA256, HMAC SHA384 ou le mode de chiffrement combiné GCM. Si la version de TLS négociée ne le permet pas, on utilisera HMAC SHA1.

3.4 Recommandations et test des implémentations

En appliquant les recommandations des encadrés précédents sur la liste des suites TLS définies par l'IANA, on obtient 51 suites acceptables sur les 297 suites définies. Parmi celles-ci, 19 sont disponibles dès la version TLSv1.0 du protocole. Ces dernières sont toutes supportées dans les versions stables d'OpenSSL (1.0.0) et NSS (3.13 avec Firefox 10), comme en témoigne le tableau de l'annexe B.

3.4.1 Test des serveurs

Il est relativement aisé de tester les suites cryptographiques supportées par les serveurs. Il suffit en effet de suivre les étapes suivantes :

1. créer la liste de suites que l'on souhaite tester ;
2. se connecter au serveur en proposant la liste de suites ;
3. si le serveur accepte la négociation,
   — noter la suite comme supportée par le serveur,
   — retirer la suite de la liste de test,
   — si la suite n'est pas vide, retourner à l'étape 2 ;
4. sinon, renvoyer la liste des suites supportées par le serveur construite jusqu'à présent.

Il existe des outils automatisant ces tests, comme `sslscan`, décrit dans l'annexe C. Bien entendu, le résultat dépend de la liste de suites données en entrée. Comme `sslscan` repose sur OpenSSL, il ne permet de tester que les suites présentes dans la bibliothèque.

Dans tous les cas, on gardera à l'esprit que les tests ci-dessus sont relativement intrusifs puisqu'ils consistent à monter puis avorter de nombreuses connexions SSL/TLS vers un serveur donné, et peuvent être perçus comme une attaque. On réservera donc ces outils pour tester des serveurs avec l'accord des administrateurs.

Voici quelques résultats sur des implémentations classiques :
— un version récente d'Apache utilisant OpenSSL ou GNUTls sous Debian accepte de négocier l'ensemble des 19 suites décrites ci-dessus (à condition qu'on lui fournisse les certificats RSA/ECDSA correspondants), ainsi que d'autres suites qui sont déconseillées ;
— il est possible de configurer Apache pour qu'il réponde uniquement avec des suites recommandées (directive `SSLCipherSuite`, voir l'annexe C) ;
— un serveur IIS sous Windows Server 2003 refuse toute suite cryptographique proposant la PFS. Parmi les suites acceptées par défaut, certaines sont recommandées, d'autre non. En particulier, la suite `RSA_RC4_MD5`, déconseillée plus haut, est celle que le serveur préfère lorsque le client la propose. Il est possible de configurer les suites acceptées à l'aide de la base de registre ;
— avec Windows Server 2008, les choses s'améliorent puisque la bibliothèque cryptographique du système supporte les courbes elliptiques, ainsi que les version 1.1 et 1.2 de TLS (avec des options de configuration). Il est donc possible d'améliorer nettement l'éventail des suites proposées.

De même, il est simple de vérifier la taille des paramètres asymétriques utilisés par un serveur en analysant le certificat présenté et les paramètres Diffie-Hellman proposés.

3.4.2 Test des clients

Côté client, les tests à mener sont plus complexes. Il est en effet nécessaire de mettre en place un grand nombre de serveurs SSL pour tester chaque suite de manière isolée, puis demander au client de tenter de se connecter sur chacun pour tester les suites réellement acceptées [10].

En effet, il ne suffit pas de regarder le `ClientHello` initial, puisque certains clients envoient un deuxième (et parfois d'autres) `ClientHello` lorsque le premier a été rejeté (voir la section 2.4), et ces messages peuvent contenir des suites cryptographiques différentes.

10. On peut également surveiller en boîte noire les `ClientHello` émis par le client pour y vérifier les suites cryptographiques proposées, mais il faut prendre en compte les mécanismes de *fallback*.

En ce qui concerne la réaction à de petits paramètres cryptographiques asymétriques, il est difficile de vérifier les limitations mises en œuvre par les clients. Deux méthodes sont possibles :
— analyser le code source, ce qui peut être fastidieux...
— demander au client de se connecter sur différents serveurs proposant de petits paramètres pour tester le comportement.

Dans tous les cas, le test en boîte noire peut s'automatiser avec `openssl s_server` et quelques lignes de *shell*.

4 Quelques réflexions sur les certificats

Cette partie décrit quelques vulnérabilités connues, et les limitations de l'infrastructure de gestion de clés utilisée classiquement dans TLS.

4.1 Validation des certificats reçus

Commençons par un rappel des vérifications à faire par le client pour s'assurer de la validité de la chaîne de certification reçue. En effet, TLS permet l'utilisation d'un IGC pour certifier les certificats terminaux des serveurs. Le client reçoit donc une suite de certificats censés représenter une chaîne de validation.

Les opérations de vérification à réaliser sont les suivantes :
— vérifier le chaînage des certificats : l'émetteur (champ `Issuer`) d'un certificat doit être le champ `Subject` du certificat suivant dans la chaîne [11] ;
— vérifier que les autorités (intermédiaires et racine) ont bien été certifiées pour ce rôle (on regardera les extensions `BasicConstraints`, `KeyUsage` et `NSCertType`) ;
— les algorithmes de signature doivent être acceptables : ils ne doivent plus reposer sur la fonction de hachage MD5, et SHA1 est aujourd'hui fortement déconseillé pour les nouveaux déploiements (voir [2] pour plus d'information) ;
— vérifier les tailles de clés (comme évoqué précédemment) ;
— valider la signature de chaque maillon ;
— vérifier les dates de validité de tous les certificats ;
— interroger les moyens de révocation pour vérifier que le certificat est toujours correct : listes de révocation (CRL), serveur en ligne (OCSP), listes noires éventuelles.
— vérifier la correspondance entre le certificat terminal et le serveur contacté. Le nom du serveur doit en effet se trouver dans le `CommonName` ou dans l'extension `SubjectAltName` ;
— vérifier que l'autorité racine dans la chaîne fait partie du magasin de confiance du client ;
— vérifier enfin toutes les extensions présentes dans les certificats et connues du client [12].

Dès qu'une des vérifications ci-dessus échoue, le client fait en pratique appel à l'utilisateur pour savoir s'il souhaite déroger aux principes de sécurité, ce qui amène généralement à un effondrement du système puisque l'utilisateur souhaitant accéder au site aura tendance à accepter la chaîne pourtant invalide.

Les sections qui suivent présentent des problèmes mettant en cause les implémentations et l'infrastructure de gestion de clés, indépendamment de l'utilisateur.

4.2 Présence de caractères nuls dans les *distinguished names*

En 2009, Moxie Marlinspike a présenté une nouvelle attaque sur les certificats utilisés dans SSL utilisant des caractères nuls dans les *distinguished names* [20].

Aujourd'hui, il est possible d'obtenir un certificat serveur SSL valide dans la majorité des navigateurs pour quelques euros seulement. Il s'agit généralement de certificats DV, pour *Domain Validation*. En effet, il existe aujourd'hui trois grandes classes de certificats, qui diffèrent par le niveau de validation utilisé par l'autorité d'enregistrement et par les contraintes imposées sur le certificat [13] :
— DV (*Domain Validation*) correspond à une simple validation que le demandeur possède bien le nom de domaine correspondant (en envoyant un mail par exemple) ;
— OV (*Organization Validation*) pour lesquels la validation passe de plus par une vérification de l'existence légale de l'organisme demandant le certificat ;
— EV (*Extended Validation*) exige des contrôles plus poussés lors de l'enregistrement, et interdit l'usage d'algorithmes cryptographiques faibles.

11. Il existe également des extensions, `AuthorityKeyIdentifier` et `SubjectKeyIdentifier`, qui rendent le chaînage non ambigu en présence d'homonymie des autorités : .
12. Si une extension marquée critique n'est pas reconnue, le certificat doit être rejeté.
13. Vu du navigateur, la différence entre les trois modes n'est pas toujours très visible, même si un effort est souvent fait pour mettre en évidence les certificats EV (barre d'URL colorée).

Dans le cas de la validation du domaine, si un client demande un certificat pour `www.sstic.org`, l'autorité enverra un courrier électronique à `root@sstic.org` pour vérifier que la demande est émise par le propriétaire du domaine.

Cependant, certaines autorités d'enregistrement utilisent des langages de programmation où les chaînes de caractères ne sont pas terminées par un caractère nul, alors que la majorité des implémentations sont codées en C et interprètent le caractère nul comme la fin de la chaîne. Fort de ce constat, Moxie Marlinspike a montré qu'il était possible d'exploiter cette incohérence dans les interprétations. Par exemple, le nom de domaine `www.sstic.org\0www.attaquant.net` était vu comme un sous domaine de `attaquant.net` par certaines autorités, alors que les navigateurs considéraient que le certificat était signé pour `www.sstic.org`.

Ainsi, la vérification du domaine était réalisée auprès de `attaquant.net`, une zone contrôlée par l'attaquant, alors que le certificat lui permettait de se faire passer pour `www.sstic.org` auprès de nombreux clients.

Les navigateurs ont été corrigés depuis pour prendre correctement en compte les `Common Name` contenant des caractères nuls invalides.

4.3 Impact de la divulgation de la clé privée d'une autorité de certification

Nous avons vu dans la section 3.1.3 l'impact lié à la divulgation de la clé privée d'un serveur TLS. Si maintenant on considère qu'un attaquant réussit à casser / récupérer la clé privée d'une autorité de certification, il lui est possible de signer des certificats X.509 de son choix (autorité intermédiaire, client TLS, serveur TLS, protection de courrier électronique S/MIME, etc.) avec cette autorité.

En particulier, si l'autorité dont la clé est compromise est reconnue par les navigateurs (soit directement, soit en tant qu'autorité intermédiaire), tous les certificats émis par l'attaquant seront considérés valides et il pourra usurper aisément tout serveur TLS à l'aide d'une attaque réseau active.

Il faut en réalité distinguer deux cas de compromission de la clé privée :

— soit l'attaquant a obtenu la clé privée en clair, auquel cas il peut réaliser toutes les opérations de signature sans aucune contrainte ;

— soit l'attaquant a accès au service de signature, par exemple suite à une intrusion dans le système d'information de l'autorité. Dans ce cas, ses possibilités de signature sont limitées par les gabarits éventuellement imposés par le service de signature. De plus, il doit maintenir son accès frauduleux à la plate-forme de signature tant qu'il souhaite émettre de nouveaux certificats.

4.4 Rappels de quelques incidents et de leurs conséquences

Deux incidents impliquant des fournisseurs de certificats importants sont survenus en 2011. En mars, une filiale de Comodo a subi une intrusion qui a permis à l'attaquant de générer plusieurs certificats de serveur TLS pour des sites importants (dont `addons.mozilla.org`, `login.live.com` ou `mail.google.com`). L'intrusion a été rapidement détectée par Comodo qui a procédé à la révocation des certificats émis et fourni la liste de ces certificats pour intégration dans les listes noires des navigateurs.

Un autre incident a été découvert au sujet de l'autorité racine Diginotar : après une intrusion en juillet 2011 dans le système d'information de l'autorité néerlandaise, des utilisateurs iraniens de Google Chrome ont détecté en août l'utilisation de certificats suspects [14] pour le site `mail.google.com`. En effet, bien que l'attaque ait été détectée par Diginotar, l'incident n'a pas été bien traité et aucune analyse des dégats n'a été réalisée avant fin août, laissant l'attaquant libre d'utiliser les certificats qu'il avait généré. Une fois l'affaire connue, les certificats racines de Diginotar ont été retirés des magasins de confiance et Diginotar a rapidement fait faillite.

Début 2012, une annonce de Trustwave a provoqué de nombreuses réactions : l'autorité de certification aurait émis en 2010 un certificat d'autorité intermédiaire à une entreprise privée qui souhaitait mettre en place une « analyse transparente du trafic » de ses employés [27]. Cette autorité aurait été révoquée depuis. Selon Trustwave, ce type de service aurait été rendu par de nombreuses autorités.

Les trois incidents décrits sont de natures différentes, mais ont le même impact : de mauvaises décisions au niveau des autorités de certification remettent en cause la sécurité de l'ensemble des clients TLS faisant confiance à ces autorités, sans qu'il n'existe de moyen simple de limiter la portée. En effet, la compromission de l'autorité de certification A peut mener à des attaques sur n'importe quel site, même si ce dernier est signé par une autorité B distincte.

La difficulté est accentuée par la taille des magasins de certificats embarqués dans les piles TLS, et le fait que dans certains cas, ils soient dynamiques (chez Microsoft et Opera par exemple, des autorités de confiance peuvent être récupérées en ligne à la demande ou lors de mises à jour, rendant difficile le suivi des autorités auxquelles on veut faire confiance).

14. Suite à l'attaque de Comodo, Google a mis en place le *certificate pinning* dans Chrome, un mécanisme permettant à certains sites de signaler quels certificats étaient acceptables pour authentifier leur site.

4.5 Quelques essais pour limiter l'impact d'une compromission

Afin de limiter l'impact d'une compromission, les moyens disponibles aujourd'hui sont très limités :
— on peut utiliser la révocation de certificats (listes de révocations, serveur OCSP), mais ces moyens sont rarement mis en place dans les faits ;
— dans certains cas, on peut radier les autorités de certification racine d'où découle la compromission, mais cela implique de rendre inaccessibles tous les sites dont les certificats remontent à l'autorité en question. De plus, de nombreuses autorités intermédiaires sont en réalité signées par plusieurs autorités racines ;
— enfin, on peut utiliser des mécanismes ad-hoc dans les clients (listes noires, *certificate pinning*), qui ont l'avantage d'être efficaces et rapidement déployés lors des mises à jour mais passent difficilement à l'échelle.

Pour améliorer la situation, plusieurs solutions ont été proposées, certaines depuis longtemps, pour rétablir la confiance dans le fonctionnement des autorités de certification utilisées par SSL/TLS.

Une des pistes est d'améliorer globalement la confiance dans les certificats, en imposant plus de contraintes aux autorités quant aux vérifications à réaliser lors de l'enregistrement et quant au traitement des incidents. C'est le sens des certificats EV et du *Certification Authority/Browser Forum*.

Des solutions techniques partielles existent :
— les listes blanches/noires, mais elles ne passent pas à l'échelle et ne concernent pas tous les clients ;
— l'utilisation de contraintes X.509 dans les certificats d'autorité. L'extension `NameConstraints` permet par exemple d'indiquer que les certificats découlant de cette autorité devront appartenir à un sous-domaine donné (`.fr` par exemple). Cependant, ces extensions sont mal interprétées par certaines piles TLS comme l'ont montré quelques tests.

D'autres solutions sont en cours de développement. On peut citer DANE (*DNS-Based Authentication of Named Entities* [4]), dont l'objectif est d'utiliser DNSSEC pour transporter de l'information sur les certificats attendus du serveur TLS. L'idée se heurte à plusieurs problèmes, le premier étant l'état du déploiement de DNSSEC à l'heure actuelle.

Le projet *Convergence* proposé par Moxie Marlinspike repose de son côté sur la vérification en ligne que le certificat vu par un navigateur est bien le même que celui vu par des témoins de confiance (les *notaries*). Il était déjà possible avec l'extension Firefox *Certificate Patrol* de vérifier dans le temps que le certificat présenté pour un site était le même, mais cette méthode provoque de nombreux faux positifs.

Plusieurs idées ont été proposées pour rendre publique la base de données des certificats valides (dont le projet *Sovereign Keys* de l'EFF). En effet, si pour accepter le certificat du serveur `www.sstic.org`, le navigateur vérifie que le certificat présenté a bien été publié ouvertement, les responsables du site peuvent également vérifier dans le temps que le seul certificat publié pour le site est bien le leur.

Enfin, une piste encore peu explorée pourrait reposer sur une meilleure expressivité dans la confiance accordée aux certificats d'autorité racine (à la manière de l'extension `NameConstraints`, mais au niveau des magasins de certificats. Il serait également souhaitable de pouvoir accepter certaines autorités en fonction de l'application utilisant le magasin de certification.

4.6 Recommandations

Comme nous l'avons vu, il n'est pas évident de répondre à la question de la confiance dans les certificats de manière générique. Dans tous les cas, la première difficulté est d'identifier les applications utilisant SSL/TLS et les magasins de certificats utilisés. Ensuite seulement il est possible de faire le tri pour ne garder que ce qui est nécessaire.

4.6.1 Côté serveur

Pour les serveurs, il est fortement recommandé de se fournir chez des prestataires de confiance, ce qui permet d'éviter de devoir changer tous les certificats en cas de compromission de l'autorité utilisée. Le gouvernement néerlandais a dû repenser en urgence la gestion de ses certificats suite à l'affaire DigiNotar, puisque l'autorité de certification nationale était opérée par ce fournisseur.

De plus, si le serveur requiert une authentification client, il faut bien limiter les magasins de confiance côté serveur à la ou les autorités réellement acceptées.

4.6.2 Côté client

Ici, le problème est plus complexe car les magasins sont parfois partagés (magasin Microsoft pour de nombreuses applications, `ca-certificates` sous Linux) et ne permettent pas d'avoir la granularité souhaitée.

En effet, dans certains cas, la politique est simple car on connaît les certificats de confiance à l'avance (SMTPS, IMAPS, LDAPS, VPN, EAP-TLS), et si le logiciel le permet, on peut (et on doit) réduire la confiance aux autorités réellement utiles.

Pour d'autres applications, ce tri est beaucoup plus dur à réaliser, par exemple le navigateur web, le client de messagerie pour S/MIME, puisqu'on ne sait pas à l'avance de quelles autorités de certification on a réellement besoin. Pour les plus courageux, et lorsque le logiciel le permet, il est possible de supprimer les autorités racines de certification du magasin, pour n'accepter que celles qui sont réellement utiles au fur et à mesure (en pratique moins de 10 ACs peuvent suffire), mais cela nécessite des vérifications à chaque mise à jour. Il est également possible de s'aider d'extensions comme *Certificate Patrol* pour évaluer le niveau de confiance à accorder à l'authentification d'un site.

Il semble donc difficile de prendre en compte complètement la menace côté client au delà de la mise à jour des logiciels. Une piste en cours d'étude est l'utilisation d'IDS pour repérer (et éventuellement bloquer) les connexions TLS mettant en jeu des certificats ouvertement révoqués. En effet, la négociation TLS étant réalisée en clair, il est possible au niveau du réseau de mettre fin à une connexion TLS authentifiée par Diginotar en inspectant le message `Certificate`, indépendamment des mises à jours des clients terminaux.

4.7 Test des implémentations

Au cours de cette étude, voici quelques éléments que nous avons obtenus concernant les magasins de certificats :

Application	Système	Type de magasin
Firefox	Linux	magasin spécifique par profil basé sur NSS
Chromium	Linux	magasin spécifique par profil basé sur NSS
Firefox/Java	Linux	magasin indépendant spécifique à Java
Chromium/Java	Linux	idem Chromium
Thunderbird	tous	magasin spécifique par profil basé sur NSS
Firefox	Windows	magasin spécifique par profil basé sur NSS
IE	Windows	magasin commun Microsoft
Firefox/Java	Windows	magasin indépendant spécifique à Java
IE/Java	Windows	magasin commun Microsoft
Outlook	Windows	magasin commun Microsoft
Apache	Linux	aucun par défaut

Comme le montre l'exemple de Java, certaines extensions embarquent un magasin autonome, qu'ils n'utilisent pas systématiquement, en fonction de l'application ou du système. Cette situation rend toute politique globale difficile à mettre en œuvre.

Concernant la révocation, le constat est clair : par défaut, aucun mécanisme n'est réellement honoré :
— les listes de révocations ne sont pas téléchargées par défaut, même lorsque l'information est disponible ;
— certains clients émettent des requêtes OCSP pour une vérification en ligne, mais en l'absence de réponse, le certificat est considéré valide.

Conclusion

L'actualité des dernières années a mis à mal SSL/TLS, que ce soit du point de vue du protocole lui-même, de certains algorithmes cryptographiques utilisés, ou encore de l'infrastructure de gestion de clés quasi-universelle sur laquelle repose traditionnellement le protocole. Cependant, des contre-mesures existent et une communauté réactive s'est emparée de la question.

Sans être complètement exhaustive, la description des attaques connues sur SSL/TLS présentée ici permet de décrire un cadre dans lequel il est possible d'utiliser correctement cette brique essentielle pour la sécurité des échanges sur internet.

Certaines recommandations sont faciles à mettre en œuvre et à tester, alors que d'autres semblent difficiles à implémenter sans le développement ou l'amélioration d'outils existants. L'abandon ces dernières années de SSLv2 par de nombreux acteurs est un signe positif, bien que très tardif. De même, le développement récent de TLSv1.1 et TLSv1.2 dans OpenSSL permettra à moyen terme d'utiliser les algorithmes cryptographiques nouvellement ajoutés à la norme, et de répondre à certaines vulnérabilités de manière plus pérenne.

De plus, des démarches sont en cours pour améliorer les navigateurs pour prendre en compte certaines vulnérabilités et relever le niveau de sécurité minimal pour une connexion HTTPS : le forum CAB, des vérifications supplémentaires faites sur l'origine des certificats ou sur la taille des paramètres cryptographiques.

Cependant, les navigateurs sont loin d'être les seuls logiciels à utiliser SSL/TLS. C'est pourquoi d'autres méthodes pourraient être mises en place pour réaliser cette surveillance de manière cohérente, homogène et indépendamment des logiciels utilisés, par exemple à l'aide de règles mises en œuvre en bordure de réseau sur un IDS/IPS. Des *patches* actuellement en cours d'intégration dans le projet Surricat vont dans ce sens.

Remerciements

Ces travaux ont été réalisés au sein de la sous-direction Assistance Conseil et Expertise de l'ANSSI dans le cadre de ma thèse à Télécom SudParis, dirigée par Hervé Debar (Télécom SudParis, département RST) et co-encadrée par Benjamin Morin (ANSSI). Je profite de ces quelques lignes pour remercier les collègues avec qui j'ai pu échanger sur ce sujet, et tout particulièrement Arnaud, Éric, Guillaume, Jean-René, Pierre et Xavier.

Références

[1] Adam Langley. Unfortunate current practices for HTTP over TLS, 2011.

[2] ANSSI. RGS Annexe B1 : Mécanismes cryptographiques, règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques, version 1.20, 2010.

[3] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064, 2012.

[4] R. Barnes. Use Cases and Requirements for DNS-Based Authentication of Named Entities (DANE). RFC 6394 (Informational), October 2011.

[5] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006. Updated by RFC 5246.

[6] 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.

[7] 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.

[8] Daniel Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *CRYPTO*, 1998.

[9] Bodo Moeller. Security of CBC Ciphersuites in SSL/TLS : Problems and Countermeasures, 2002-2004.

[10] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[11] 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.

[12] 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.

[13] D. Eastlake 3rd. Transport Layer Security (TLS) Extensions : Extension Definitions. RFC 6066 (Proposed Standard), January 2011.

[14] Entrust. Entrust Bulletin on Certificates Issued with Weak 512-bit RSA Keys by Digicert Malaysia, 2011.

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

[16] Kipp E.B. Hickman. The SSL Protocol, 1994-1995.

[17] IANA. Transport Layer Security (TLS) Parameters, 2005-2012.

[18] B. Kaliski. PKCS #1 : RSA Encryption Version 1.5. RFC 2313 (Informational), March 1998. Obsoleted by RFC 2437.

[19] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, K., Emmanuel Thomé, Joppe Bos, W., Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, L., Dag Arne Osvik, Herman Te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350, Santa Barbara, United States, 2010. Springer Verlag. The original publication is available at www.springerlink.com.

[20] Moxie Marlinspike. More Tricks For Defeating SSL In Practice, 2009.

[21] Mozilla. Bug 583337 - Firefox detects, won't work with, server doing SSL DHE cipher suites with tiny keys, 2010-2011.

[22] NIST. FIPS 186-2 Digital Signature Standard, 2000.

[23] NIST. FIPS 186-3 Digital Signature Standard, 2000.

[24] M. Ray. Authentication Gap in TLS Renegotiation, 2009.

[25] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.

[26] ticalc.org. All ti signing keys factored, 2009.

[27] Trustwave. Clarifying The Trustwave CA Policy Update, 2012.

[28] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), March 2011.

[29] T. Zoller. TLS/SSLv3 renegotiation vulnerability explained, 2009-2011.

A Détection des versions du protocole supportées

Pour détecter les versions et suites cryptographiques supportées par un client ou un serveur, un moyen simple est d'utiliser les outils `s_client` et `s_server` disponibles dans la boîte à outils `openssl`. Cependant, pour pouvoir tester toutes les versions, depuis SSLv2 (maintenant désactivée par défaut dans les paquets Debian) jusqu'à TLSv1.2 (uniquement dans la branche de développement du projet) et un maximum de suites cryptographiques, il est nécessaire d'utiliser une version récente (1.0.1 au minimum) et de la recompiler.

Le plus simple sous Debian est de récupérer la version 1.0.1 et :

```
apt-get -o Apt::Default-Release=sid source openssl
cd openssl-1.0.1
```

Ensuite, on lance la compilation, en activant quelques options (on remplace `no-idea` par `enable-idea` et `no-ssl2` par `enable-ssl2` dans la ligne `CONFARGS` du fichier `debian/rules`) :

```
vi debian/rules
debuild -us -uc -rfakeroot
cd ..
sudo dpkg -i *.deb
```

A.1 Test des serveurs

Dans ce sens, il suffit d'essayer de se connecter avec `openssl s_client` vers le serveur à tester en imposant la version à tester. Afin de réduire les faux négatifs, il peut être intéressant d'ajouter l'option `-cipher ALL`.

Si le serveur à tester est `www.sstic.org`, on fera donc :

```
for version in ssl2 ssl3 tls1 tls1_1 tls1_2; do
  echo -n "$version "
  openssl s_client -cipher ALL -connect www.sstic.org:443 -$version \
            < /dev/null &> /dev/null || echo -n "not "
  echo supported
done
```

Dans ce cas, seules les versions SSLv3 et TLSv1.0 sont acceptées et mènent à une négociation réussie :

```
ssl2 not supported
ssl3 supported
tls1 supported
tls1_1 not supported
tls1_2 not supported
```

A.2 Test des clients

Ici, il va tout d'abord falloir générer un certificat pour notre serveur de test. Pour cela, on pourra simplement générer un bi-clé RSA avec la commande suivante :

```
openssl req -x509 -newkey rsa:2048 -out cert.pem
```

On obtient un certificat auto-signé et une clé privée (fichiers `cert.pem` et `privkey.pem`) qui vont nous servir pour tester nos clients. Pour les besoins du test, on ouvrira un profil vierge dans le navigateur à tester et on importera le certificat `cert.pem` le temps du test.

Ensuite, il faut démarrer un serveur SSL par version du protocole dans des consoles différentes :

```
openssl s_server -cipher ALL -cert cert.pem -key privkey.pem -ssl2 -www -accept 4000
openssl s_server -cipher ALL -cert cert.pem -key privkey.pem -ssl3 -www -accept 4001
openssl s_server -cipher ALL -cert cert.pem -key privkey.pem -tls1 -www -accept 4002
openssl s_server -cipher ALL -cert cert.pem -key privkey.pem -tls1_1 -www -accept 4003
openssl s_server -cipher ALL -cert cert.pem -key privkey.pem -tls1_2 -www -accept 4004
```

Il suffit alors de tenter une connexion vers les différents ports ouverts (4000 à 4004) pour tester le support des différentes versions. Si le navigateur affiche une page d'informations sur la connexion en cours, c'est qu'il a accepté la version du protocole. Sinon, vous aurez un autre message d'erreur et `s_server` fera apparaître des lignes d'erreur, par exemple, pour une connexion SSLv3 avortée :

```
ERROR
139945043793576:error:1408F10B:SSL routines:SSL3_GET_RECORD:wrong version number:s3_pkt.c:339:
shutting down SSL
CONNECTION CLOSED
```

A.3 Analyse de quelques implémentations

Voici un récapitulatif des serveurs testés avec les options par défaut (la colonne 5746 traite de l'implémentation de la renégociation sécurisée) :

Serveur	SSL2	SSL3	TLS1.0	TLS1.1	TLS1.2	5746
Apache 2.2.16 Debian (`mod_ssl`)		oui	oui			oui
Apache 2.2.22 Debian (`mod_ssl`)		oui	oui	oui	oui	oui
Apache 2.* Debian (`mod_gnutls`)		oui	oui	oui	oui	oui
IIS 7.5		oui	oui			oui

et quelques informations sur les clients (avec les options par défaut) :

Client	SSL2	SSL3	TLS1.0	TLS1.1	TLS1.2	5746
FF 3.5 sous Debian		oui	oui			oui
FF 3.5 sous Windows		oui	oui			oui
FF 9 sous Debian		oui	oui			oui
FF 9 sous Windows		oui	oui			oui
Chromium 16 sous Debian		oui	oui			oui
Webkit 1.6.3 (GNUTls 2.12.18) sous Debian		oui	oui	oui	oui	oui
IE8 sous XP		oui	oui			oui
IE9 sous 7		oui	oui			oui
Opera 9 sous Debian			oui			oui
Androïd 1.5		oui	oui			
Safari sur iOS 5.1		oui	oui	oui	oui	oui

B Liste des suites cryptographiques acceptables

La liste des suites cryptographiques conformes aux recommandations de la section 3 est récapitulée dans le tableau ci-dessous. La colonne TLS1 indique si la suite en question est compatible avec TLSv1.0. La colonne « Nom OpenSSL » donne le nom sous lequel la suite existe dans OpenSSL lorsqu'elle est présente. Les trois dernières colonnes indiquent les suites présentes dans les différentes versions d'OpenSSL. Il est à noter que NSS 3.13 (dans Firefox 10) implémente les mêmes suites que OpenSSL 1.0.0 par défaut [15].

Suite cryptographique	TLS1	Nom OpenSSL	0.9.8	1.0.0	1.0.1
TLS_RSA_WITH_RC4_128_SHA	oui	RC4-SHA	oui	oui	oui
TLS_RSA_WITH_3DES_EDE_CBC_SHA	oui	DES-CBC3-SHA	oui	oui	oui
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	oui	EDH-RSA-DES-CBC3-SHA	oui	oui	oui
TLS_RSA_WITH_AES_128_CBC_SHA	oui	AES128-SHA	oui	oui	oui
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	oui	DHE-RSA-AES128-SHA	oui	oui	oui
TLS_RSA_WITH_AES_256_CBC_SHA	oui	AES256-SHA	oui	oui	oui
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	oui	DHE-RSA-AES256-SHA	oui	oui	oui
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	oui	CAMELLIA128-SHA		oui	oui
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	oui	DHE-RSA-CAMELLIA128-SHA		oui	oui
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	oui	CAMELLIA256-SHA		oui	oui
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	oui	DHE-RSA-CAMELLIA256-SHA		oui	oui
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	oui	ECDHE-ECDSA-RC4-SHA		oui	oui
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	oui	ECDHE-ECDSA-DES-CBC3-SHA		oui	oui
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	oui	ECDHE-ECDSA-AES128-SHA		oui	oui
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	oui	ECDHE-ECDSA-AES256-SHA		oui	oui
TLS_ECDHE_RSA_WITH_RC4_128_SHA	oui	ECDHE-RSA-RC4-SHA		oui	oui
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	oui	ECDHE-RSA-DES-CBC3-SHA		oui	oui
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	oui	ECDHE-RSA-AES128-SHA		oui	oui
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	oui	ECDHE-RSA-AES256-SHA		oui	oui
TLS_RSA_WITH_AES_128_CBC_SHA256		AES128-SHA256			oui
TLS_RSA_WITH_AES_256_CBC_SHA256		AES256-SHA256			oui
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256		DHE-RSA-AES128-SHA256			oui
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256		DHE-RSA-AES256-SHA256			oui
TLS_RSA_WITH_AES_128_GCM_SHA256		AES128-GCM-SHA256			oui
TLS_RSA_WITH_AES_256_GCM_SHA384		AES256-GCM-SHA384			oui
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256		DHE-RSA-AES128-GCM-SHA256			oui
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384		DHE-RSA-AES256-GCM-SHA384			oui
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256		ECDHE-ECDSA-AES128-SHA256			oui
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384		ECDHE-ECDSA-AES256-SHA384			oui
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256		ECDHE-RSA-AES128-SHA256			oui
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384		ECDHE-RSA-AES256-SHA384			oui
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256		ECDHE-ECDSA-AES128-GCM-SHA256			oui
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384		ECDHE-ECDSA-AES256-GCM-SHA384			oui
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256		ECDHE-RSA-AES128-GCM-SHA256			oui
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384		ECDHE-RSA-AES256-GCM-SHA384			oui
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256					
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256					
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256					
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256					
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256					
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384					
TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256					
TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384					
TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256					
TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384					
TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256					
TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384					
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256					
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384					
TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256					
TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384					

15. En revanche, NSS 3.13 avec Firefox 3.5 semble ne pas accepter ECDSA comme algorithme de signature.

C Détection des suites cryptographiques supportées par un serveur avec sslscan

Pour tester rapidement les suites cryptographiques supportées par un serveur donné, on peut utiliser `sslscan`. Par exemple, la commande suivante affiche l'ensemble des suites acceptées par le site `www.sstic.org`[16] :

```
sslscan www.sstic.org | grep Accepted
```

La liste des suites est la suivante et est la même quel que soit le protocole effectivement négocié (SSLv3 ou TLSv1.0) :

Suite cryptographique	Taille des clés de chiffrement
DHE-RSA-AES256-SHA	256
AES256-SHA	256
EDH-RSA-DES-CBC3-SHA	168
DES-CBC3-SHA	168
DHE-RSA-AES128-SHA	128
AES128-SHA	128
RC4-SHA	128
RC4-MD5	128

Parmi les suites proposées, les sept premières sont acceptables au vu des recommandations de ce document. Cependant, le site accepte de négocier la suite RC4-MD5 qu'il est conseillé de refuser. On peut également regretter que le serveur n'accepte pas les suites cryptographiques utilisant ECDHE.

C.1 Avertissements

L'utilisation de cet outil pose cependant quelques soucis.

Tout d'abord, la démarche est assez intrusive puisque le client `sslscan` monte plusieurs dizaines de sessions TLS en quelques secondes avec le serveur, ce qui peut être perçu comme une attaque.

Par ailleurs, même avec une version d'OpenSSL récente, `sslscan` ne sait pas utiliser TLSv1.1 ni TLSv1.2 pour le moment. Pour le site `www.sstic.org`, ce n'est pas grave puisque ces protocoles n'étaient pas supportés.

C.2 Quelques éléments sur la gestion des suites dans OpenSSL

Terminons cette annexe par un exemple de configuration Apache sur les suites cryptographiques. Supposons que `httpd.conf` contient une ligne de la forme :

```
SSLCipherSuite  HIGH:RC4
```

En effet, l'administrateur de ce serveur veut s'assurer que les suites cryptographiques utilisent des clés assez longues (`HIGH`). Cependant, comme il veut également offrir les suites utilisant RC4, il a ajouté `RC4`.

Cela peut sembler évident à qui a déjà joué avec la sous-commande `ciphers` d'`openssl`, mais en ajoutant ce mot clé, l'administrateur a en réalité ajouté *toutes* les suites utilisant `RC4`, y compris les suites d'export et les suites sans authentification.

Pour le vérifier, on peut lancer `sslscan` ou simplement tester la liste des suites avec la commande suivante[17] :

```
% openssl ciphers -v HIGH:RC4
ADH-AES256-SHA          SSLv3 Kx=DH       Au=None Enc=AES(256)  Mac=SHA1
DHE-RSA-AES256-SHA      SSLv3 Kx=DH       Au=RSA  Enc=AES(256)  Mac=SHA1
DHE-DSS-AES256-SHA      SSLv3 Kx=DH       Au=DSS  Enc=AES(256)  Mac=SHA1
AES256-SHA              SSLv3 Kx=RSA      Au=RSA  Enc=AES(256)  Mac=SHA1
ADH-AES128-SHA          SSLv3 Kx=DH       Au=None Enc=AES(128)  Mac=SHA1
DHE-RSA-AES128-SHA      SSLv3 Kx=DH       Au=RSA  Enc=AES(128)  Mac=SHA1
DHE-DSS-AES128-SHA      SSLv3 Kx=DH       Au=DSS  Enc=AES(128)  Mac=SHA1
AES128-SHA              SSLv3 Kx=RSA      Au=RSA  Enc=AES(128)  Mac=SHA1
ADH-DES-CBC3-SHA        SSLv3 Kx=DH       Au=None Enc=3DES(168) Mac=SHA1
EDH-RSA-DES-CBC3-SHA    SSLv3 Kx=DH       Au=RSA  Enc=3DES(168) Mac=SHA1
EDH-DSS-DES-CBC3-SHA    SSLv3 Kx=DH       Au=DSS  Enc=3DES(168) Mac=SHA1
DES-CBC3-SHA            SSLv3 Kx=RSA      Au=RSA  Enc=3DES(168) Mac=SHA1
DES-CBC3-MD5            SSLv2 Kx=RSA      Au=RSA  Enc=3DES(168) Mac=MD5
ADH-RC4-MD5             SSLv3 Kx=DH       Au=None Enc=RC4(128)  Mac=MD5
EXP-ADH-RC4-MD5         SSLv3 Kx=DH(512)  Au=None Enc=RC4(40)   Mac=MD5  export
```

16. Bien entendu, les tests réalisés sur ce serveur ont été menés avec l'accord de l'administrateur.
17. Le lecteur attentif remarquera que les noms utilisés par OpenSSL ne sont pas les noms officiels des suites.

```
RC4-SHA                 SSLv3 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=SHA1
RC4-MD5                 SSLv3 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=MD5
EXP-RC4-MD5             SSLv3 Kx=RSA(512) Au=RSA  Enc=RC4(40)   Mac=MD5  export
EXP-RC4-MD5             SSLv2 Kx=RSA(512) Au=RSA  Enc=RC4(40)   Mac=MD5  export
RC4-MD5                 SSLv2 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=MD5
```

Ainsi, il est essentiel de *toujours* vérifier l'ensemble des listes sélectionnées lorsqu'on configure OpenSSL sur un serveur ou un client. On laisse le lecteur curieux tester les ensembles suivants :

— `ALL`
— `COMPLEMENTOFALL`
— `HIGH:RC4:-EXPORT`
— `HIGH:-EXPORT:RC4`
— `HIGH:!EXPORT:RC4`

Il remarquera en particulier que l'ordre compte puisqu'OpenSSL permet en fait de définir un ensemble *ordonné* de suites cryptographiques.