RFC 7525 : Recommendations for Secure Use of TLS and DTLS

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 6 mai 2015

Date de publication du RFC : Mai 2015

https://www.bortzmeyer.org/7525.html

Le protocole de sécurité TLS sécurise aujourd'hui, par la cryptographie, un grand nombre de communications par l'Internet. Le protocole et ses mises en œuvre dans des bibliothèques comme OpenSSL ont eu plusieurs failles, réparées au fur et à mesure. Mais les plus grosses failles restent dans l'utilisation pratique de TLS : programmeurs et administrateurs systèmes font des erreurs qui affaiblissent la sécurité de TLS. Ce RFC, le deuxième du groupe UTA <https://tools.ietf.org/wg/uta> (utilisation de TLS par les applications) résume les bonnes pratiques qui devraient être suivies en matière de sécurité pour que TLS atteigne complètement ses objectifs. (Il a depuis été remplacé par le RFC 9325 [1].)

TLS est normalisé dans le RFC 5246. Ce RFC documente la version 1.2 et la 1.3 est en cours de normalisation. Il est surtout connu pour sa sécurisation de HTTP, nommée HTTPS. Mais il sert aussi pour sécuriser SMTP, XMPP, IMAP, etc. TLS lui-même ne marche que sur TCP mais il a un équivalent pour UDP, nommé DTLS, et normalisé dans le RFC 6347. Si un attaquant veut casser la protection qu'offre TLS (par exemple pour accéder au contenu en clair d'une communication normalement chiffrée), il peut viser (le RFC 7457 donne une bonne liste) :

— Le protocole lui-même. Ainsi, SSL v3, un précurseur de TLS, avait une faille de conception qui permettait l'attaque Poodle. Les failles du protocole, par définition, sont indépendantes d'une mise en œuvre particulière de TLS.
— Les algorithmes cryptographiques utilisés. TLS a la propriété d'« agilité cryptographique » qui fait qu'il n'est pas lié à un algorithme particulier mais offre un choix. Cela permet, au fur et à mesure des progrès de la cryptanalyse, de changer d'algorithme. Par exemple, les algorithmes AES-CBC (RFC 3602) et RC4 (RFC 7465) ont tous les deux montré des faiblesses sérieuses, ce qui fait que leur utilisation dans TLS est découragée. La liste des algorithmes possibles dépend de la bibliothèque TLS utilisée mais aussi de la configuration du client et du serveur TLS. Pour retirer un algorithme désormais trop faible, comme RC4, on peut donc agir dans le logiciel (en retirant le code correspondant) ou dans la configuration, en supprimant l'algorithme de la liste acceptée.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc9325.txt

— TLS est mis en œuvre dans des bibliothèques logicielles qui, comme tous les logiciels, ont des bogues. Cette fois-ci la faille est spécifique à une mise en œuvre donnée et la correction est de mettre à jour le logiciel, une fois la version corrigée sortie. Ce fut le cas pour la faille Heartbleed, spécifique à OpenSSL, ou de la faille goto fail, spécifique à Apple.
— Même si le protocole est parfait, n'utilisant que des algorithmes cryptographiques incassés, et programmé sans une bogue, des failles de sécurité peuvent quand même surgir si TLS est utilisé imprudemment. Par exemple, si une application utilise TLS en ayant débrayé (ou pas activé) la vérification du certificat, TLS peut être contourné via une attaque de l'Homme du Milieu. L'article « "*The most dangerous code in the world : validating SSL certificates in non-browser software*" <http://doi.acm.org/10.1145/2382196.2382204> » en donne plusieurs exemples.

Bref, l'analyse de sécurité de TLS est complexe et on voit pourquoi il faut se méfier comme de la peste des résumés journalistiques qui annoncent une faille TLS alors qu'il ne s'agissait que d'une bogue dans une bibliothèque.

Le RFC 7457 décrivait déjà les failles, avec plein d'exemples. Notre nouveau RFC se concentre sur les recommandations. Il définit un plancher de sécurité (pas un plafond : on peut toujours être plus strict que ce que définit le RFC). D'autre part, le RFC prévient évidemment que ces recommandations évolueront avec le temps.

Le cœur de notre RFC n'est donc pas l'analyse des risques, mais les recommandations pratiques (sections 3 et suivantes). D'abord, sur la version du protocole :

— Ne jamais utiliser SSL, que ce soit la v2 (RFC 6176) ou la v3. Décrite dans le RFC 6101, cette dernière a encore bien trop de trous (comme celui qui permet POODLE) et ne gère pas des extensions essentielles à la sécurité comme celle du RFC 5746.
— N'utiliser TLS 1.0 (RFC 2246) que si on n'a pas le choix. Publié il y a seize ans (!), il n'accepte pas certains algorithmes de cryptographie moderne et il a d'autres faiblesses.
— TLS 1.2 doit être privilégié. Notons que les algorithmes cryptographiques recommandés dans notre RFC ne sont disponibles qu'avec cette version 1.2.

Si un client TLS a un mécanisme de repli (on tente de se connecter avec la version N, ça échoue, on réessaie avec N-1, ce qui se nomme la "*downgrade dance*"), on ne doit jamais se replier vers SSL. Autrement, un Homme du Milieu aurait juste à faire échouer la connexion initiale, pour garantir l'utilisation d'un protocole faible (notez que le RFC 7507 fournit une solution contre ce repli). Aux dernières nouvelles, seuls 3 % des serveurs Web TLS/SSL de l'Internet étaient SSL-seulement.

Lorsque client et serveur essaient de se connecter avec TLS mais acceptent des connexions sans TLS, des attaques actives peuvent forcer un repli vers la connexion sans TLS, en clair et pas sécurisée du tout (la négociation de TLS n'est pas elle-même protégée par TLS). Pour éviter cela, client et serveur devraient insister sur l'utilisation de TLS et utiliser des techniques comme HSTS (RFC 6797) pour signaler qu'on sait faire du TLS et qu'il ne faut pas essayer en clair.

Plusieurs attaques sont facilitées par l'utilisation de la compression (RFC 5246, section 6.2.2), comme CRIME. Il est recommandé de la couper, sauf si on est sûr que le protocole applicatif qui utilise TLS n'est pas vulnérable à ces attaques.

L'établissement d'une session TLS nécessite d'échanger plusieurs paquets, ce qui prend du temps <https://www.bortzmeyer.org/latence.html>. Pour limiter ce problème, TLS dispose d'un mécanisme de reprise de session (RFC 5246). Ce mécanisme doit être autant protégé que le serait une session normale. Par exemple, dans le cas du RFC 5077, si on chiffrait le ticket utilisé pour la reprise de session avec un algorithme bien plus faible que celui utilisé dans la session elle-même, un attaquant pourrait choisir de tenter une « reprise » d'une session inexistante, court-circuitant ainsi l'authentification. À noter aussi que ce mécanisme de reprise de session peut annuler l'effet de la "*forward secrecy*" : si l'attaquant met la main sur les tickets, il peut déchiffrer les sessions qu'il a enregistré (cf. section 3.4 de notre RFC). Il faut donc changer les clés des tickets de temps en temps.

L'extension de renégociation sécurisée du RFC 5746 doit être disponible (ou, sinon, il faut couper la renégociation).

Une dernière recommandation générale porte sur SNI (RFC 6066, section 3). Bien que cela n'affecte pas directement la sécurité, cette extension est recommandée, pour des protocoles comme HTTPS, car elle permet des politiques de sécurité plus souples et donc plus implémentables (comme d'avoir un certificat par site Web).

La section 3 de notre RFC portait sur des recommandations générales. La section 4 s'attaque aux algorithmes de cryptographie ("*cipher suites*"). Ceux-ci s'usent avec le temps, quand les progrès de la cryptanalyse exposent leurs failles. Cette usure se fait à un rythme très variable : certains algorithmes durent depuis longtemps, d'autres ont été éliminés rapidement. Mais aucun algorithme n'est éternel et voilà pourquoi il est nécessaire de séparer le protocole des algorithmes, ce qu'on nomme l'« agilité cryptographique ». Notre RFC recommande donc de ne pas utiliser l'algorithme `NULL`, qui ne chiffre pas du tout (pas besoin de cryptanalyse pour le casser...) et n'a d'intérêt que pour les tests et le débogage <https://www.bortzmeyer.org/crypto-debug.html>. Plus récemment, les progrès de la cryptanalyse, documentés dans le RFC 7465, ont montré que RC4 ne devait **pas** être utilisé non plus. Le problème des algorithmes trop faibles est aggravé par l'action des États qui veulent pouvoir espionner les communications et qui imposent, dans certains cas, l'utilisation d'algorithmes dont les faiblesses sont connues. C'est le cas aux États-Unis des algorithmes dit "*export*" (car ils étaient les seuls légalement exportables à une époque). Si ces algorithmes sont présents, l'attaque FREAK permettra à un attaquant actif de les sélectionner, flanquant en l'air toute la sécurité de TLS.

Les révélations de Snowden ont montré que la NSA (et peut-être d'autres organisations) enregistre de nombreuses communications chiffrées qu'elle ne sait pas décrypter. L'idée est que si, un jour, on met la main sur la clé privée (par perquisition physique, piratage de l'ordinateur à distance ou tout autre méthode), on pourra alors déchiffrer et certaines de ces communications seront encore intéressantes. Pour éviter cela, notre RFC conseille de privilégier les algorithmes qui mettent en œuvre la "*forward secrecy*" comme la famille DHE.

Quels algorithmes restent-ils après ces nombreuses éliminations ? Notre RFC en mentionne quatre (après une discussion longue et acharnée à l'IETF) : `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256`, `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`, `TLS_DHE_RSA_WITH_AES_256_GCM_SHA384` et `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`. Ils utilisent DHE ou ECDHE, avec RSA, AES en mode GCM (voir RFC 5288 section 6, sur la sécurité de ce mode) et SHA-2. Ils font tous du chiffrement intègre ("*authenticated encryption*", RFC 5116). Tout logiciel TLS devrait être configuré pour préférer ces algorithmes (cf. « "*Applied Crypto Hardening*" <https://bettercrypto.org/static/applied-crypto-hardening.pdf> »).

Attention, notre RFC parle uniquement de préférer ces algorithmes, pas de les imposer. Pour permettre l'interopérabilité, TLS a en effet des algorithmes obligatoires, qui doivent être mis en œuvre dans chaque bibliothèque TLS. C'est le cas de `TLS_RSA_WITH_AES_128_CBC_SHA`, obligatoire mais pas le plus fort, loin de là. Mais on peut compter sur lui puisqu'il est obligatoire, contrairement aux quatre algorithmes cités plus haut, qui peuvent ne pas être présents. Il peut donc être nécessaire d'arbitrer entre interopérabilité et sécurité...

Ce RFC a également des recommandations sur les clés publiques utilisées dans les certificats : 2 048 bits au moins en RSA (ce qui est imposé par la plupart des AC depuis un certain temps, c'est le cas de CAcert <https://www.bortzmeyer.org/cacert.html>, par exemple), et SHA-256 pour condenser la clé (et plus SHA-1 <https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/2-R4XziFc7A/YO0ZSrX_X4wJ>).

Un petit mot en passant sur un point abordé dans la section 5. Celle-ci concerne l'« applicabilité » des recommandations faites ici. Cette applicabilité est très large, quasiment tous les usages de TLS sont concernés. Il existe un usage fréquent de TLS, parfois appelé « chiffrement opportuniste » (RFC 7435) où une des parties qui communiquent va essayer de faire du TLS mais, si cela ne marche pas (par exemple parce que l'autre partie ne gère pas le chiffrement), on se rabat sur une communication en clair. Dans ce contexte, certaines des recommandations du RFC peuvent être trop strictes. Par exemple, on a vu qu'il ne fallait pas utiliser RC4, trop faible cryptographiquement. Si on prend au pied de la lettre cette recommandation, qu'une des deux parties ne connait que RC4 et qu'on est dans un contexte de chiffrement opportuniste, on pourrait arriver à ce résultat paradoxal que, RC4 étant refusé, on communique en clair, ce qui est certainement bien pire que le plus mauvais des algorithmes de cryptographie. Il peut donc être nécessaire de dévier des recommandations de notre RFC 7525.

Enfin, pour terminer ce RFC déjà riche, une section 6 consacrée à divers problèmes de sécurité qui n'avaient pas été traités avant. D'abord, la nécessité, lorsqu'on vérifie un certificat, de vérifier également que le nom du serveur, dans le certificat, corespond au nom auquel on voulait se connecter. Cela parait évident mais ce n'est pas toujours fait par la bibliothèque qu'on utilise (cf. mon exposé à Devoxx <http://blog.soat.fr/2014/04/devoxx-2014-utiliser-tls-sans-se-tromper-une-conference-animee> - et ses transparents (en ligne sur https://www.bortzmeyer.org/files/bortzmeyer-tls-devoxx.odp) - pour des exemples.) Il peut donc être utile que l'application fasse la validation elle-même, sans quoi un attaquant actif peut détourner le trafic chiffré, de manière triviale. La section 3 du RFC 2818 et le RFC 6125 détaillent ce qu'il faut exactement valider. Bien sûr, pour que cette validation ait un sens, il faut que le nom du serveur ait lui-même été obtenu de manière sûre. Si on a cliqué sur un lien https://... dans un message de hameçonnage, vérifier que le nom est correct n'aide pas tellement (« oui, c'est bien secure-serious-bank.ab451e239f.ru »). Idem si le nom vient d'une requête DNS (par exemple de type MX) non sécurisée par DNSSEC.

J'ai déjà parlé plus haut de la "*forward secrecy*", une propriété qui fait que le vol des clés privées a posteriori est inutile, même pour un attaquant qui avait enregistré les communications chiffrées. Dans la vie, ce vol peut arriver pour des tas de raisons, machines jetées à la poubelle sans les avoir effacées, et récupérées par un indiscret, clés trop faibles face à la force brute (cf. « "*Mining Your Ps and Qs : Detection of Widespread Weak Keys in Network Devices*" <https://factorable.net/paper.html> »), clé générée par un tiers (dans le « "*cloud*" ») et ensuite obtenue par piratage, attaque violente et/ou légale contre les administrateurs système pour obtenir la clé, et toutes les autres méthodes d'accès aux clés privées qui peuvent exister. La "*forward secrecy*" n'empêche pas ces attaques mais elle empêche l'attaquant d'exploiter son succès. Elle s'obtient en général en faisant un échange Diffie-Hellman. Dans le contexte de la sécurité de TLS, le principal problème est que beaucoup d'algorithmes utilisés ne permettent pas le "*forward secrecy*", et que les autres, les bons, ne sont pas sélectionnés en priorité.

L'authentification du serveur TLS (on peut aussi authentifier le client mais c'est plus rare) repose presque toujours sur un certificat X.509. Si la clé privée associée au certificat est volée, le mécanisme de sécurité de X.509 est la révocation (RFC 5280, section 3.3). On sait que ce mécanisme nécessite une bonne connexion Internet, et est peu fiable : les listes de certificats révoqués, à distribuer partout, ne passent pas à l'échelle, le protocole OCSP (RFC 6960) est souvent bloqué (et il pose de toute façon des problèmes de passage à l'échelle et de vie privée), menant les clients à ignorer les échecs OCSP, etc. Il existe des solutions partielles (section 8 du RFC 6066, RFC 6961, RFC 6698) mais pas assez déployées.

Si vous voulez lire des recommandations en français sur la configuration de TLS, il y a deux guides ANSSI qui sont en rapport avec ce RFC, le RGS <https://www.ssi.gouv.fr/entreprise/guide/cryptographie-les-regles-du-rgs/> pour les algorithmes de cryptographie, et le guide pour la sécurité des sites Web <https://www.ssi.gouv.fr/entreprise/guide/recommandations-pour-la-secu> , avec des recommandations précises pour OpenSSL (SSLCipherSuites Apache).

Et pour terminer sur une note personnelle, si vous regardez la configuration HTTPS de mon blog, elle est actuellement perturbée par deux sérieuses bogues du module GnuTLS d'Apache : #754960 <https://bugs.debian.org/754960> et #642357 <https://bugs.debian.org/642357>.

Merci à Manuel Pégourié-Gonnard pour une relecture attentive qui a permis de trouver plusieurs erreurs. D'autres bonnes lectures :

— *"Applied Crypto Hardening"* <https://bettercrypto.org/static/applied-crypto-hardening.pdf>
— *"Mozilla SSL [sic] Configuration Generator"* <https://mozilla.github.io/server-side-tls/ssl-config-generator/>