PRISE EN MAIN DE TLS 1.3
AVEC OPENSSL 1.1.1

OLIVIER LEVILLAIN
[Maître de conférences à Télécom SudParis]

MOTS-CLÉS : SSL/TLS / TLS 1.3 / OPENSSL

TLS 1.3, la dernière version du protocole SSL/TLS, a été standardisée en août dernier. Dans la foulée, la version 1.1.1 de la boîte à outils OpenSSL, qui inclut le support de TLS 1.3, a été publiée. Cet article propose un aperçu de TLS 1.3 et de son utilisation avec OpenSSL 1.1.1.

SSL/TLS est sans doute le protocole de sécurité le plus utilisé sur Internet pour protéger les communications. Depuis 2011, le protocole a beaucoup fait parler de lui suite aux nombreuses failles de sécurité publiées (le lecteur intéressé pourra consulter deux publications faites au SSTIC en 2012 et 2015 **[1][2]**). TLS 1.3 (RFC 8446), la dernière mouture du protocole sortie à l'été dernier, répond à ces problèmes et propose des communications plus sécurisées... et plus rapides. Avant d'étudier les outils en ligne de commande, commençons par un bref rappel du protocole et de son histoire.

1. BREF HISTORIQUE DE SSL/TLS

Il est d'usage de parler de SSL/TLS lorsqu'on s'intéresse au protocole de sécurité bien connu. En effet, la première version publique du protocole était SSLv2, publiée en 1994 par **Netscape**, pour permettre la protection des communications web. La création du schéma d'URL `https://`

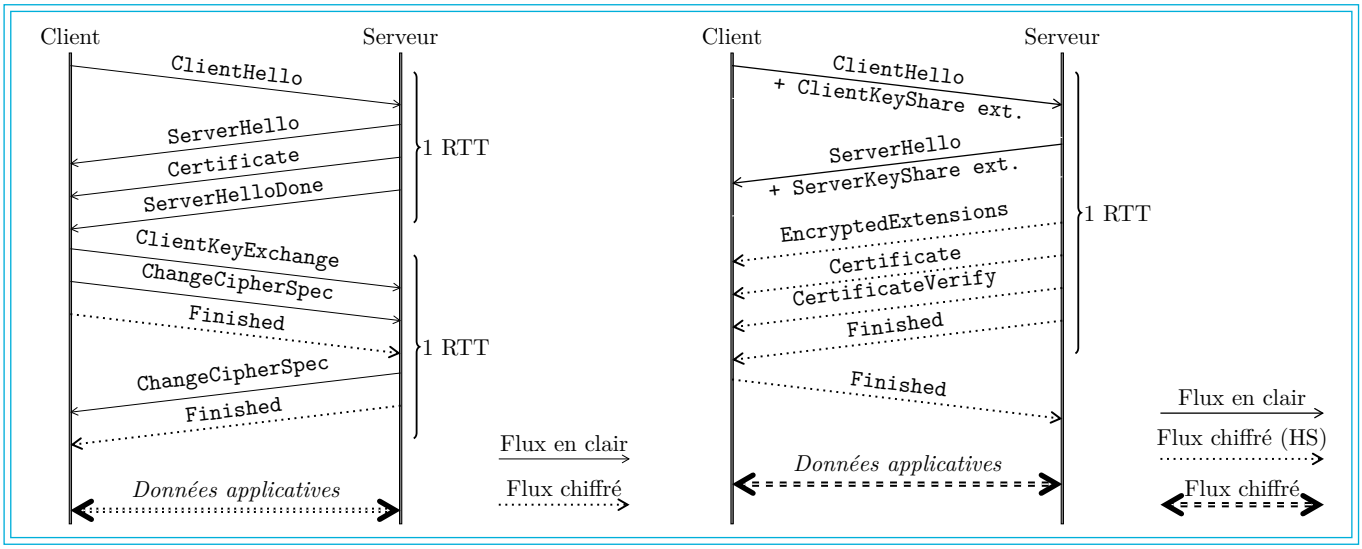

Fig. 1 : Comparaison de l'établissement de session TLS avec SSLv3-TLS 1.2 (à gauche) et l'établissement de session avec TLS 1.3 (à droite).

permettait en effet l'authentification du serveur, ainsi que la protection en confidentialité et en intégrité des données échangées, autant de propriétés essentielles à l'échange de données sensibles, comme un numéro de carte bancaire !

Cependant, SSLv2 a rapidement montré des faiblesses structurelles importantes, obligeant les acteurs du Web à proposer une révision du protocole... ou plutôt deux ! SSLv3 pour Netscape et PCT 1.0 pour **Microsoft**... Les deux protocoles corrigeaient les défauts principaux de SSLv2, mais c'est SSLv3 de Netscape qui a été retenu finalement pour devenir un standard de l'Internet. L'IETF a ainsi publié la RFC 2246, qui décrivait TLS 1.0, très proche de SSLv3.

Si l'Histoire a essentiellement oublié PCT, certains racontent que l'existence d'un concurrent aurait conduit l'IETF à renommer SSL pour contenter les deux partis. Il en résulte que TLS 1.0 est le successeur de SSLv3, ce qui n'est pas forcément évident pour tout le monde, si on ne regarde que les chiffres. Dans le reste de cet article, il ne sera néanmoins question que de TLS, les deux versions de SSL étant aujourd'hui jugées obsolètes.

Initialement conçu pour protéger les communications HTTP, TLS est aujourd'hui devenu la solution quasi- universelle pour sécuriser les communications réseau. On le retrouve ainsi naturellement dans le monde du Web, mais également dans de nombreux autres protocoles, par exemple dans la messagerie électronique avec IMAP, POP ou SMTP. Par conception, TLS s'intercale en effet très facilement entre TCP et presque n'importe quel protocole applicatif.

Au-delà de cet usage classique, on retrouve également le protocole de négociation de TLS (le *Handshake Protocol*) dans des outils comme **OpenVPN** ou dans le schéma d'authentification extensible EAP, notamment utilisé dans WPA. Dans les deux cas, TLS permet d'établir un secret partagé, qui est ensuite utilisé dans d'autres protocoles.

Cette omniprésence amène naturellement à se demander si TLS est à la hauteur pour garantir la sécurité d'Internet.

2. TLS 1.3

Entre 2014 et 2018, le groupe de travail TLS de l'IETF a planché sur la nouvelle version du protocole. Cette révision avait deux objectifs principaux : dépoussiérer TLS pour y intégrer des constructions cryptographiques à l'état de l'art, d'une part et rendre TLS plus rapide, d'autre part.

Pour arriver à ce résultat, TLS a en pratique été repensé en profondeur, avec une négociation par défaut en 1 RTT (*round-trip time*) ; cela signifie qu'avant que le client puisse émettre des données applicatives, il suffit d'un aller-retour avec le serveur (contre deux auparavant). La figure 1 compare l'établissement d'une session TLS avant et après TLS 1.3.

Avec les versions précédentes, le client initie la connexion avec le message `ClientHello` dans lequel il annonce les algorithmes et fonctionnalités qu'il supporte. Le serveur choisit les paramètres qu'il retient pour la session, puis

présente son certificat. Un second aller-retour sert ensuite à l'échange de clés. C'est seulement après que les données peuvent être échangées.

Dans le mode par défaut de TLS 1.3, l'échange de clés est réalisé avec les extensions **ClientKeyShare** et **ServerKeyShare** (incluses respectivement dans les messages **ClientHello** et **ServerHello**), ce qui permet concrètement de réaliser la négociation des paramètres, l'échange de clés et l'authentification du serveur avec un aller-retour de moins.

NOTE

Évidemment, les détails sont un peu plus compliqués. Tout d'abord, il faut noter que les allers-retours dont il est question ici sont les échanges de segments TCP, qui ne tiennent pas compte de l'établissement de session TCP (le *3-way handshake*). Ensuite, dans certains cas, la négociation en 1 RTT peut échouer et nécessiter un aller-retour supplémentaire pour négocier les paramètres de l'échange de clés ; ce cas devrait cependant être rare entre des implémentations classiques utilisant des configurations standardisées.

2.1 Un mot sur la reprise de session

Il existe en pratique un mécanisme de reprise de session, présent dans toutes les versions de TLS, qui permet de monter rapidement une session sans réauthentifier le serveur ni dérouler l'échange de clés. Dans toutes les versions précédentes de TLS, ce mode permettait de monter des sessions TLS avec un serveur connu en 1 RTT.

Avec TLS 1.3, le mécanisme perdure, mais inclut désormais une nouvelle fonctionnalité : la possibilité d'émettre des données dès la première série de messages, juste après le **ClientHello**. Ce mode 0 RTT permet d'accélérer encore l'envoi de données utilisateur, mais au prix de garanties de sécurité amoindries : le serveur doit être prêt à gérer le rejeu de données 0 RTT.

Ces modes, qui seront testés un peu plus loin, reposent sur le stockage d'une session composée du matériel cryptographique généré lors de la session initiale. Le client pourra ainsi revenir voir le serveur en proposant de repartir de cette session précédente.

3. PREMIERS PAS AVEC OPENSSL 1.1.1

Pour pouvoir jouer avec TLS 1.3, il va falloir utiliser une version récente d'OpenSSL. Pour cet article, la version 1.1.1a du 20 novembre 2018, présente dans la version *unstable* de **Debian**, a été utilisée. Il est bien entendu également possible d'installer OpenSSL depuis les sources **[3]**.

3.1 Installation et vérifications

Une fois OpenSSL 1.1.1 installé, on peut vérifier le numéro de version à l'aide de la commande **version** :

```
$ openssl version
OpenSSL 1.1.1a  20 Nov 2018
```

On peut aussi vérifier la présence du support pour TLS 1.3 dans l'outil en invoquant la commande **s_client** et en demandant de l'aide :

```
$ openssl s_client -help
Usage: s_client [options]
Valid options are:
[...]
-tls1_3         Just use TLSv1.3
[...]
```

Une fois ces vérifications faites, il est possible de tester une connexion TLS 1.3 avec un serveur sur Internet. On peut utiliser par exemple https://tls13.crypto.mozilla.org/ qui a l'intérêt de n'accepter les connexions qu'avec la dernière version du protocole. L'appel avec la commande OpenSSL **s_client** se fait de la manière suivante :

```
$ openssl s_client -connect tls13.
crypto.mozilla.org:443
```

Si tout se passe bien, vous devriez voir dans un premier temps des informations sur les certificats présentés par le serveur :

```
[...]
Certificate chain
 0 s:CN = tls13.crypto.mozilla.org
   i:C = US, O = Let's Encrypt, CN =
Let's Encrypt Authority X3
 1 s:C = US, O = Let's Encrypt, CN =
Let's Encrypt Authority X3
   i:O = Digital Signature Trust Co.,
CN = DST Root CA X3
[...]
```

Ensuite, **s_client** vous donne des informations sur les paramètres négociés, dont voici quelques morceaux choisis :

```
New, TLSv1.3, Cipher is TLS_AES_128_
GCM_SHA256
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
```

Après avoir affiché toutes ces informations, OpenSSL vous permet d'échanger des données qui seront transportées par le canal sécurisé. Si vous souhaitez communiquer avec le serveur de test de **Mozilla**, il faudra cependant parler un dialecte HTTP/1.1 parfait : cela implique d'inclure l'en-tête **Host** et de signaler les fins de ligne par **\r\n** et non par **\n** (on peut pour cela utiliser l'option **-crlf** de **s_client**).

3.2 Capture de la négociation

Relançons à présent la commande en capturant le trafic en parallèle à l'aide de **tcpdump** (en tant que root) :

```
# tcpdump -w /tmp/capture.pcap port 443
```

Depuis un autre terminal :

```
% openssl s_client -connect tls13.
crypto.mozilla.org:443
```

On peut ensuite étudier la capture à l'aide de **Wireshark** ou pour ceux qui préfèrent la ligne de commande, à l'aide de **tshark** :

```
$ tshark -2 -R ssl -r /tmp/capture.pcap
    1   0.187762 192.168.0.180 →
52.32.149.186 TLSv1.3 382  Client Hello
    2   0.390023 52.32.149.186 →
192.168.0.180 TLSv1.3 1514 Server Hello,
Change Cipher Spec
    3   0.391890 52.32.149.186 →
192.168.0.180 TLSv1.3 230  Application Data
    4   0.392867 192.168.0.180 →
52.32.149.186 TLSv1.3 130  Change Cipher
Spec, Application Data
    5   0.584879 52.32.149.186 →
192.168.0.180 TLSv1.3 309  Application Data
```

On retrouve bien nos premiers messages, comme annoncé plus haut (**ClientHello** et **ServerHello**), mais le reste des messages est uniquement constitué de messages **Change Cipher Spec** et **Application Data**.

Le premier type de message (**Change Cipher Spec**) est un archaïsme optionnel hérité des versions précédentes de TLS. Il s'agit en réalité d'un message factice, dont l'unique fonction est de faire ressembler TLS 1.3 à TLS 1.2. De cette manière, les équipements réseau présents sur le chemin (il peut s'agir de pare-feu ou de proxys filtrants, que l'on regroupe parfois sous la dénomination de *middleboxes*) ont moins de risque de couper la connexion. Certains espèrent qu'un jour, les équipements réseau comprendront TLS 1.3 et que cette verrue au protocole deviendra inutile.

L'autre type de message est en fait un type générique qui encapsule tous les messages chiffrés, y compris les messages suivants de la négociation.

À ce stade, on peut donc observer la négociation initiale (soit en ajoutant **-V** à **tshark**, soit en cliquant sur les messages **Hello** dans Wireshark). Étudions dans un premier temps la négociation de la version TLS.

Jusqu'à TLS 1.2, le client devait annoncer dans le champ **Version** la plus haute version du protocole qu'il supportait. Cependant, ce mécanisme n'a jamais vraiment été compris par certains éditeurs de logiciels ou fabricants d'équipements réseau, qui rejetaient un message avec un champ **Version** trop élevé (au lieu de simplement répondre avec la version maximale qu'ils supportaient, comme indiqué dans la spécification). Pour cette raison, le groupe de travail TLS de l'IETF a choisi de figer le champ **Version** à la version 1.2 et d'utiliser une nouvelle extension (**supported_versions**) pour annoncer les versions supportées (ici TLS 1.2 et TLS 1.3). C'est également via cette extension que le serveur sélectionne la version qui sera utilisée pour le reste de l'échange.

```
# Côté client
Extension: supported_versions (len=5)
    Type: supported_versions (43)
    Length: 5
    Supported Versions length: 4
    Supported Version: TLS 1.3 (0x0304)
    Supported Version: TLS 1.2 (0x0303)

# Côté serveur
Extension: supported_versions (len=2)
    Type: supported_versions (43)
    Length: 2
    Supported Version: TLS 1.3 (0x0304)
```

Concentrons-nous maintenant sur les suites cryptographiques (*Cipher Suites*). Le client OpenSSL en propose 31 dans l'exemple :

```
Cipher Suites (31 suites)
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    [...]
    Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)
```

Là encore, il y a du changement avec la nouvelle version du protocole ! Les anciennes suites décrivaient l'ensemble des algorithmes nécessaires à l'établissement de session (échange de clé, authentification, chiffrement, motif d'intégrité), alors que TLS 1.3 négocie les algorithmes dans trois endroits différents : l'extension **supported_groups** pour l'échange de clé, l'extension **signature_algorithms** pour l'authentification du serveur et les suites cryptographiques pour la protection des messages à l'aide d'un algorithme de chiffrement authentifié (AEAD, pour *Authenticated Encryption with Associated Data*).

Les trois premières suites de la liste sont donc des suites TLS 1.3, utilisant par exemple AES en mode GCM pour la première. Ensuite, la liste contient un grand nombre de suites cryptographiques compatibles avec les versions précédentes de TLS et la liste se termine par une suite factice, **TLS_EMPTY_RENEGOTIATION_INFO_SCSV**, qui correspond à une extension de sécurité (la renégociation sécurisée, RFC 5746, devenue inutile avec TLS 1.3).

Nous pourrions continuer de détailler les messages **Hello**, mais il semble également intéressant de creuser ce qui se passe dans les autres messages, ceux qui sont protégés par le chiffrement authentifié. Pour cela, il faut demander à **s_client** de transmettre les secrets cryptographiques à Wireshark.

3.3 Analyse des messages et du trafic applicatif

Depuis 2010, la bibliothèque NSS (*Network Security Services*, la bibliothèque cryptographique utilisée par les produits Mozilla) implémente un mécanisme de *debug* nommé **SSL_KEYLOGFILE**. Son fonctionnement est le suivant : si une variable d'environnement nommée **SSL_KEYLOGFILE** est présente, NSS l'interprétera comme un nom de fichier où écrire les secrets cryptographiques générés, lors de l'établissement des sessions TLS.

ATTENTION !

SSL_KEYLOGFILE est un outil très pratique pour décapsuler les flux protégés par TLS et comprendre certains problèmes subtils. Il s'agit cependant aussi d'une faille de sécurité potentielle, si elle est utilisée par un attaquant. Ainsi, il est essentiel de ne pas activer ce mécanisme en production. Sous **Linux**, on peut détecter son utilisation avec un simple **grep SSL_KEYLOGFILE /proc/*/environ**.

À partir de sa version 1.1.1, OpenSSL propose également un mécanisme similaire dans ses outils de test en ligne de commande (**s_client** et **s_server**). Il s'agit de l'option **-keylogfile**. Relançons donc la capture précédente en ajoutant ce paramètre à notre client TLS :

```
% openssl s_client -connect tls13.crypto.mozilla.org:443
                  -keylogfile /tmp/secrets.txt -crlf
CONNECTED(00000003)
[...]
---
read R BLOCK
GET / HTTP/1.1
Host: tls13.crypto.mozilla.org:443
```

Nous avons également envoyé une requête HTTP dans **s_client**, ce qui provoque une longue réponse de la part du serveur, que nous allons essayer de retrouver dans Wireshark. Pour cela, jetons d'abord un coup d'œil au fichier généré par l'option **-keylogfile** :

```
# SSL/TLS secrets log file, generated by OpenSSL
SERVER_HANDSHAKE_TRAFFIC_SECRET a8aa6c... 249a00...
EXPORTER_SECRET a8aa6c... 48e5c9...
SERVER_TRAFFIC_SECRET_0 a8aa6c... 4245d0...
CLIENT_HANDSHAKE_TRAFFIC_SECRET a8aa6c... a98a45...
CLIENT_TRAFFIC_SECRET_0 a8aa6c... cdc149...
```

Chaque ligne non commentée décrit un secret (**CLIENT_TRAFFIC_SECRET_0** pour le secret de session concernant les communications du client vers le serveur), indexé par une valeur (ici **a8aa6c**...) qui correspond à la valeur aléatoire publiée par le client dans son premier message (**ClientHello.Random**).

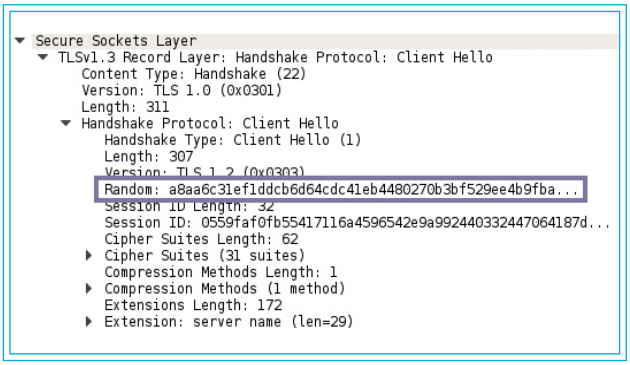

Fig. 2 : Mise en évidence du champ Random émis par le client dans l'échange capturé.

Lorsqu'on ouvre la capture dans Wireshark, on peut tout d'abord constater que le champ **Random** contient bien la valeur qui a servi d'indice dans le fichier **secrets.txt** (voir figure 2).

Ensuite, ouvrons les options des protocoles (menu ***Edit / Preferences***). On trouve les options de déchiffrement dans la liste de gauche, dans l'item ***Protocols***, puis SSL. Il suffit alors de remplir le champ ***(Pre)-Master-Secret log filename*** comme indiqué sur la figure 3.

La simple application de la configuration fait apparaître dans la fenêtre de capture le nom des messages de la négociation qui nous étaient masqués jusqu'à présent, par exemple **Encrypted Extensions**, **Certificate** ou **Finished**. De plus, la connexion étant établie sur le port **443**, Wireshark interprète le flux clair auquel il a maintenant accès comme du HTTP. Il devient dès lors possible

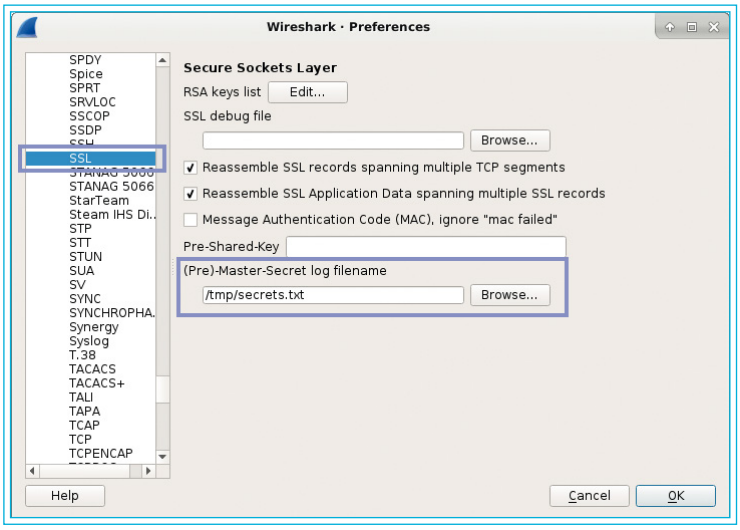

Fig. 3 : Mise en évidence des options permettant le déchiffrement TLS.

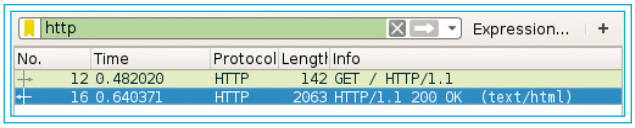

Fig. 4 : Filtrage avec le mot-clé http pour isoler le dialogue HTTP.

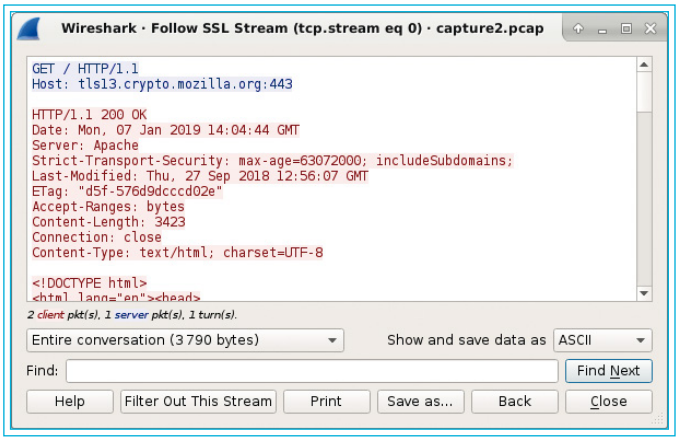

Fig. 5 : Contenu du flux en clair, obtenu avec la fonctionnalité Follow SSL Stream.

de filtrer le dialogue HTTP (voir figure 4) ou de récupérer l'ensemble du transcript en clair, via l'option ***Follow SSL Stream*** du menu contextuel (voir figure 5), qui rappelle évidemment le bel outil qu'est ***Follow TCP Stream***.

4. TESTS DE PERFORMANCE

Afin de mettre en évidence le temps d'établissement de session avec TLS 1.3, nous allons mettre en place un serveur web simpliste avec la commande **s_server** d'OpenSSL.

On suppose que l'on dispose d'une machine **C**, depuis laquelle nous lancerons le client HTTPS et d'une machine **S**, sur laquelle tournera le serveur HTTPS en utilisant le port **4433**.

4.1 Génération d'un certificat autosigné

Pour monter un tel service, nous allons avoir besoin d'un certificat X.509. Pour les besoins de nos tests, nous allons utiliser un simple certificat autosigné. Bien entendu, pour un déploiement en production, il sera nécessaire d'utiliser des

certificats issus d'une infrastructure de gestion de clés (par exemple, en utilisant une autorité de certification commerciale ou un service comme *Let's Encrypt* **[4]**).

La commande suivante génère une clé privée RSA (**key.pem**) et produit un certificat autosigné contenant la clé publique (**cert.pem**) :

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

4.2 Évaluation du RTT

Afin de déterminer la durée d'un aller-retour réseau entre le client **C** et le serveur **S**, le plus simple est de s'assurer que le port **4433** n'est pas filtré sur la machine **S**, mais qu'aucun service n'écoute sur le port. Une tentative de connexion générera immédiatement un **RESET** de la part de **S**. Sous Linux, on peut avoir le même comportement en ajoutant temporairement une règle **iptables REJECT** :

```
# iptables -A INPUT -p tcp --dport 4433
--state NEW -j REJECT
```

NOTE

Pour supprimer la règle ensuite, il suffit de taper la même commande en remplaçant **-A** (*add*) par **-D** (*delete*).

Ensuite, il suffit de tenter de se connecter sur **S** depuis **C** et de mesurer le temps mis par la connexion pour échouer. Le plus simple pour cela est d'utiliser **netcat** (le paquetage éponyme fournit la commande **nc**) et de mesurer le temps avec **time** :

```
% time nc {Adresse S} 4433
(Nom de la machine S) [Adresse S] 4433 (?) :
Connection refused

real 0m0.035s
user 0m0.000s
sys  0m0.000s
```

La valeur qui nous intéresse est **real**, c'est-à-dire le temps réel écoulé pendant l'exécution du programme (les deux autres lignes donnent le temps CPU consommé pendant l'exécution du programme, en espace utilisateur et dans le noyau).

Afin de produire une mesure moins bruitée, on peut exécuter le script **bash** suivant pour faire plusieurs connexions et moyenner le temps obtenu :

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

echo $(( $total / $N ))
```

Dans le test réalisé, la valeur de 34 millisecondes représente de manière assez précise le temps d'un aller-retour réseau. Évidemment, cette valeur peut varier d'un environnement à l'autre.

4.3 Comparaison du temps d'établissement d'une session TLS 1.2

Sur le serveur **S**, nous lançons à présent un serveur TLS en utilisant les fichiers créés plus haut (l'option **www** indique à **s_server** de se comporter comme un serveur HTTPS rudimentaire) :

```
% openssl s_server -accept 4433 -cert
cert.pem -key key.pem -www
```

Côté client, on peut alors mesurer le temps d'une connexion avec la commande suivante :

```
% echo "GET /" | ( time openssl s_client
-connect {Adresse S}:4433 ) 2&>1 | grep real
real 0m0.119s
```

En réalisant plusieurs mesures avec un script similaire à celui utilisé précédemment, on obtient 124 millisecondes en moyenne.

Pour tester le comportement de TLS 1.2, il suffit d'ajouter l'option **-tls1_2** sur la ligne de commande de **s_client** pour forcer la version à utiliser. Une mesure de l'exécution de cette commande donne en moyenne 155 millisecondes.

Il apparaît donc un écart de 31 millisecondes entre l'utilisation de TLS 1.2 et TLS 1.3 du point de vue du client, ce qui est cohérent avec la mesure d'un aller-retour.

Pendant l'exécution de ces commandes, il peut être intéressant de capturer le trafic, pour s'assurer des échanges

réellement observés. Avec TLS 1.2, on peut alors mettre en évidence 4 RTT : un pour l'établissement de la session TCP, deux pour l'établissement de la session, un pour la transmission de la requête et de la réponse applicative. Avec TLS 1.3, on observe en revanche uniquement 3 RTT (un de moins pour l'établissement de session TLS).

4.4 Impact de la reprise de session dans TLS 1.2

Nous pouvons maintenant activer la reprise de session avec TLS 1.2. Elle est activée par défaut côté serveur. Pour l'activer côté client, il nous faut d'abord sauvegarder les informations de session TLS. Ensuite, nous pouvons mesurer l'établissement de session avec reprise de session :

```
% openssl s_client -connect
{Adresse S}:4433 -tls1_2
    -sess_out sessions-tls12

% echo "GET /" | ( time
openssl s_client -sess_in
sessions-tls12

-connect {Adresse S}:4433 )
2&>1 | grep real
real 0m0.124s
```

Ainsi, en activant la reprise de session dans TLS 1.2, le temps complet de la connexion tombe à 3 RTT, comme pour TLS 1.3, ce qui était le résultat attendu.

4.5 Activation du mode 0 RTT

Si on active simplement la reprise de session avec TLS 1.3, il n'y aura pas de gain significatif, puisque l'établissement de session TLS se fera toujours en 1 RTT. La seule accélération proviendra dans ce cas de l'absence de calculs cryptographiques asymétriques, dont la durée est négligeable devant le temps de transmission sur le réseau.

En revanche, il est possible avec TLS 1.3 d'activer une autre forme de reprise de session, le mode 0 RTT, qui permet au client d'émettre des données applicatives juste après le **ClientHello**. Pour cela, il faut relancer le serveur avec l'option **-early_data** (mais sans **-www**, car les options sont incompatibles) :

```
% openssl s_server -accept 4433 -cert cert.pem -key key.pem -early_data
```

Ensuite, il faut établir une première session TLS 1.3 en sauvegardant les informations :

```
% sleep 1 | openssl s_client -connect {Adresse S}:4433
                              -sess_out sessions sessions-tls13
```

Contrairement à ce qui a été fait pour TLS 1.2, il est ici nécessaire d'attendre un peu avant de clore la session, car les tickets de session (qui servent à la reprise de session) sont envoyés après la fin de la négociation. C'est pourquoi nous avons ajouté un appel à **sleep** avant de clore l'entrée standard.

Il ne nous reste plus qu'à lancer la connexion avec des données 0 RTT :

```
% echo Blabla > early_data.txt
% openssl s_client -connect {Adresse S}:4433
                    -sess_in sessions sessions-tls13
                    -keylogfile secrets.txt
                    -early_data early_data.txt
```

En capturant l'échange avec **tcpdump** et en ouvrant le fichier **.pcap** avec Wireshark comme précédemment, on peut remarquer plusieurs choses. Tout d'abord, il existe un message **Application Data** supplémentaire dans la première fournée de messages du client (après le **ClientHello** et le message factice **ChangeCipherSpec**) : il s'agit des fameuses données 0 RTT (ou *early data*).

Ensuite, si vous fournissez l'accès aux secrets à Wireshark comme nous l'avons fait plus haut, vous pouvez sélectionner le paquet contenant ce message supplémentaire et cliquer tout en bas sur l'onglet ***Decrypted SSL*** (voir figure 6, page suivante).

5. APPORTS DE TLS 1.3 À LA SÉCURITÉ

Un des objectifs initiaux de TLS 1.3 était d'améliorer la sécurité de TLS, suite aux nombreuses failles découvertes ces dernières années. La nouvelle spécification a effectivement été l'occasion de dépoussiérer le protocole.

Tout d'abord, seules les constructions cryptographiques robustes ont été conservées. Voici quelques exemples d'algorithmes ou de modes qui ont disparu

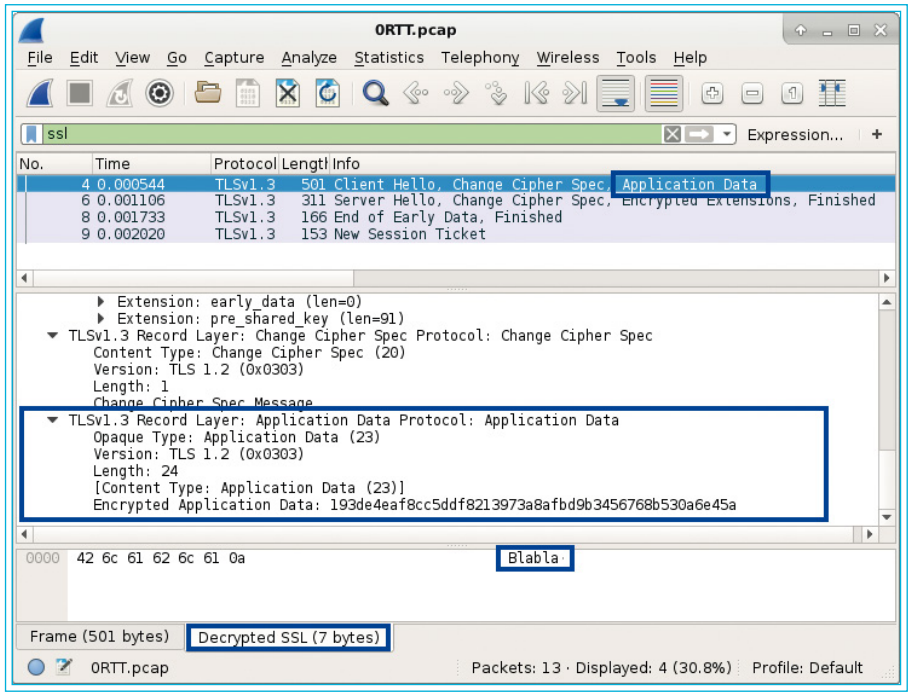

Fig. 6 : Contenu des données 0 RTT en clair, obtenu dans l'onglet Decrypted SSL.

avec la nouvelle version : l'échange de clés par chiffrement RSA, l'algorithme de chiffrement RC4, les fonctions de hachage MD5 et SHA-1.

Ensuite, la cinématique du protocole a été repensée pour que la machine à état soit plus simple et plus robuste.

Enfin, TLS 1.3 a été étudié par de nombreuses équipes de recherche, tout au long de la phase de spécification, afin de prouver formellement des propriétés de sécurité du protocole. Ce travail a permis d'obtenir des garanties fortes sur TLS 1.3.

Il existe cependant plusieurs limitations importantes.

- L'**implémentation** : même si la spécification offre aujourd'hui de bonnes garanties de sécurité, les outils mettant en œuvre TLS 1.3 doivent être développés de manière sécurisée pour que ces propriétés soient réellement apportées par ces outils.
- Le **modèle de sécurité pour l'authentification serveur** : dans TLS, l'authentification du serveur repose sur des certificats X.509. La confiance dans le protocole découle donc logiquement de la qualité de l'infrastructure de clés utilisée et donc, du sérieux des acteurs impliqués.
- Le **poids de l'histoire** : il faudra encore vivre avec les versions précédentes de TLS pendant encore plusieurs années (voire décennies). Même si des mécanismes pour contrer la négociation à la baisse existent, la cohabitation avec les versions obsolètes est à prendre en compte dans l'analyse globale des risques.
- Le **mode 0 RTT** : il s'agit d'une fonctionnalité complexe de TLS 1.3 qui n'offre pas les mêmes propriétés de sécurité que les autres modes du protocole. Là encore, c'est un élément à prendre en compte dans l'évaluation des risques.

La sécurité de TLS 1.3 est évidemment un sujet qui mériterait un article complet, ce que vous pourrez trouver dans un numéro de *MISC* à venir.

CONCLUSION

TLS 1.3 est une évolution assez importante du standard, qui apporte de belles choses, tant sur le plan de la sécurité que sur le plan des performances. Cet article vous a proposé un premier aperçu de ce qu'était TLS 1.3 et comment l'observer avec des outils classiques comme OpenSSL et Wireshark. ■

REMERCIEMENTS

Merci à Jean-Sylvain pour sa relecture attentive.

RÉFÉRENCES

[1] O. LEVILLAIN, *« SSL/TLS : état des lieux et recommandations »* : https://www.sstic.org/2012/presentation/ssl_tls_soa_recos/

[2] O. LEVILLAIN, *« SSL/TLS : 3 ans plus tard »* : https://www.sstic.org/2015/presentation/ssltls_soa_reloaded/

[3] Dépôt GitHub officiel d'OpenSSL : https://github.com/openssl/openssl

[4] Site officiel de Let's Encrypt **:** https://letsencrypt.org/