

PROJECT SR2I208

Analysis and experimentation of TLS 1.3

OUZINEB Sohaïb-ASSOMANY Marvin-PROIETTI Harith
Supervised by: Ahmed Serhrouchni

Summary:

0 - Introduction - Historical development

I - Operation of TLS

1) Handshake TLS1.2

2) TLS 1.3

a) From TLS 1.2 to TLS 1.3

b) Session initialization (4 steps)

c) Connection recovery (TLS 1.2 and TLS 1.3)

d) 0-RTT

3) Management of cryptographic services

a) Management of authentication

b) Verification of certificates

c) Management of confidentiality

d) Management of integrity

e) Management of non-replay

II - Benchmarking

1) Introduction

2) Connection time

3) Modeling of several clients

III - Extensions Client Hello

1) Extension replay protection

2) Extension rejected reason

Conclusion

Bibliography

Summary:

In this report, we analyze the contributions of TLS 1.3 compared to TLS 1.2 by precisely analyzing the handshake of session initialization, session resumption and 0-RTT by highlighting the role of each field.

We then propose a mid-scale benchmarking to evaluate the connection time and compare it to TLS 1.2. Then, we model multiple clients making multiple queries across different streams to test performance at mid-scale.

Finally, we analyze the weaknesses of 0-RTT and offer extensions to the Hello client to remedy them: one to indicate the anti-replay policy and another to indicate the value associated with this policy. We also offer another extension so that the customer can know the reason for rejection of 0-RTT to prevent a more general replay attack.

Introduction:

TLS (Transport Layer Security) is an encryption and security protocol for Internet exchanges, successor to the now obsolete SSL (Secure Socket Layer) protocol. SSL, which was created in 1994 by the Netscape organization, gave rise to 3 successive versions between 1994 and 1997, before the third SSL 3.0 version was taken over by the IETF and replaced in 1999 by TLS (Transport Layer Security).

Currently, the name TLS, SSL or even SSL / TLS is used interchangeably to designate this encryption protocol widely used by current communications. Note that TLS is largely inspired by SSL 3.0, although improvements in particular in terms of security have been made.

Currently, the version of TLS used is TLS 1.3, the RFC of which was published in August 2018. This latest version marks in particular the disappearance of certain cryptographic primitives deemed too weak in terms of security (MD4, RC4, DSA or SHA-224). In addition, the “handshake” becomes better optimized than in previous versions, and the configuration of TLS 1.3 is easier. We also note the appearance of the “0-RTT” option, which makes it possible to immediately start a communication between a client and a server having already communicated before.

The clear advantage of SSL / TLS is its ergonomics: indeed, it integrates very well with the protocols commonly used on the net, in particular HTTP, which only needs to be

modified very little to allow connection to be established. with TLS. The latter makes it possible to make the connection between the transport layer and the application layer. Currently, it is considered to be part of the session level (level 5) of the OSI model. The merger between HTTP and SSL gives rise to the well-known HTTPS protocol, operating on port 443.

Currently, TLS 1.3 is used on many servers (Apache, NGINX...) and browsers (Chrome, Safari, Opera...), its operation in making an optimized protocol while being secure.

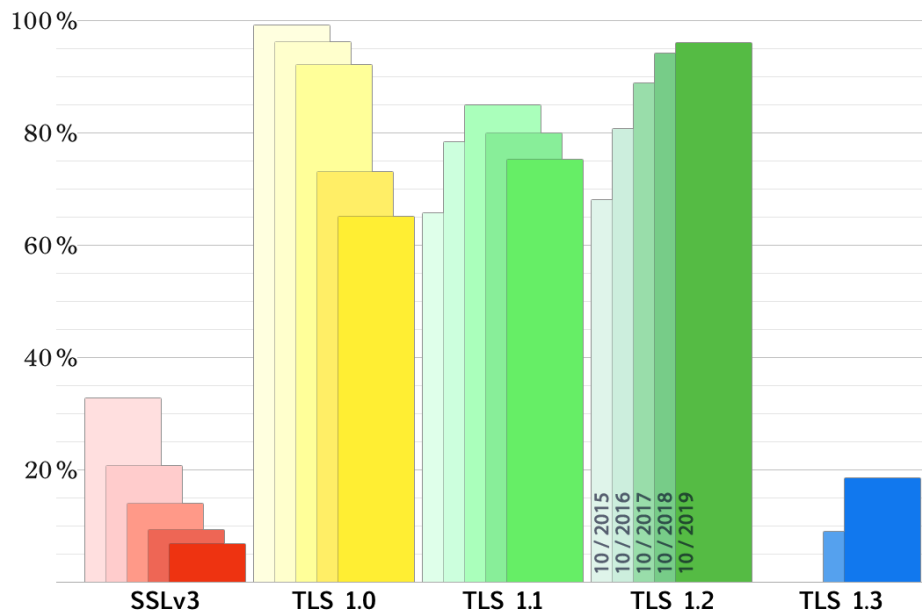


Figure: Evolution of SSL / TLS versions accepted by servers scanned by SSL Pulse over the period 2015-2018

I Operation of TLS

1) Handshake up to TLS 1.2

The TLS protocol is based on a client-server architecture. During the establishment of the connection, the client and the server send each other requests accompanied by information (certificates, shared secret keys, etc.) in a necessary step called a “handshake”. This initiation of communication will ensure that the main cryptographic services will be respected during future exchanges. The main mechanisms to be respected are authentication, confidentiality, integrity and non-replay.

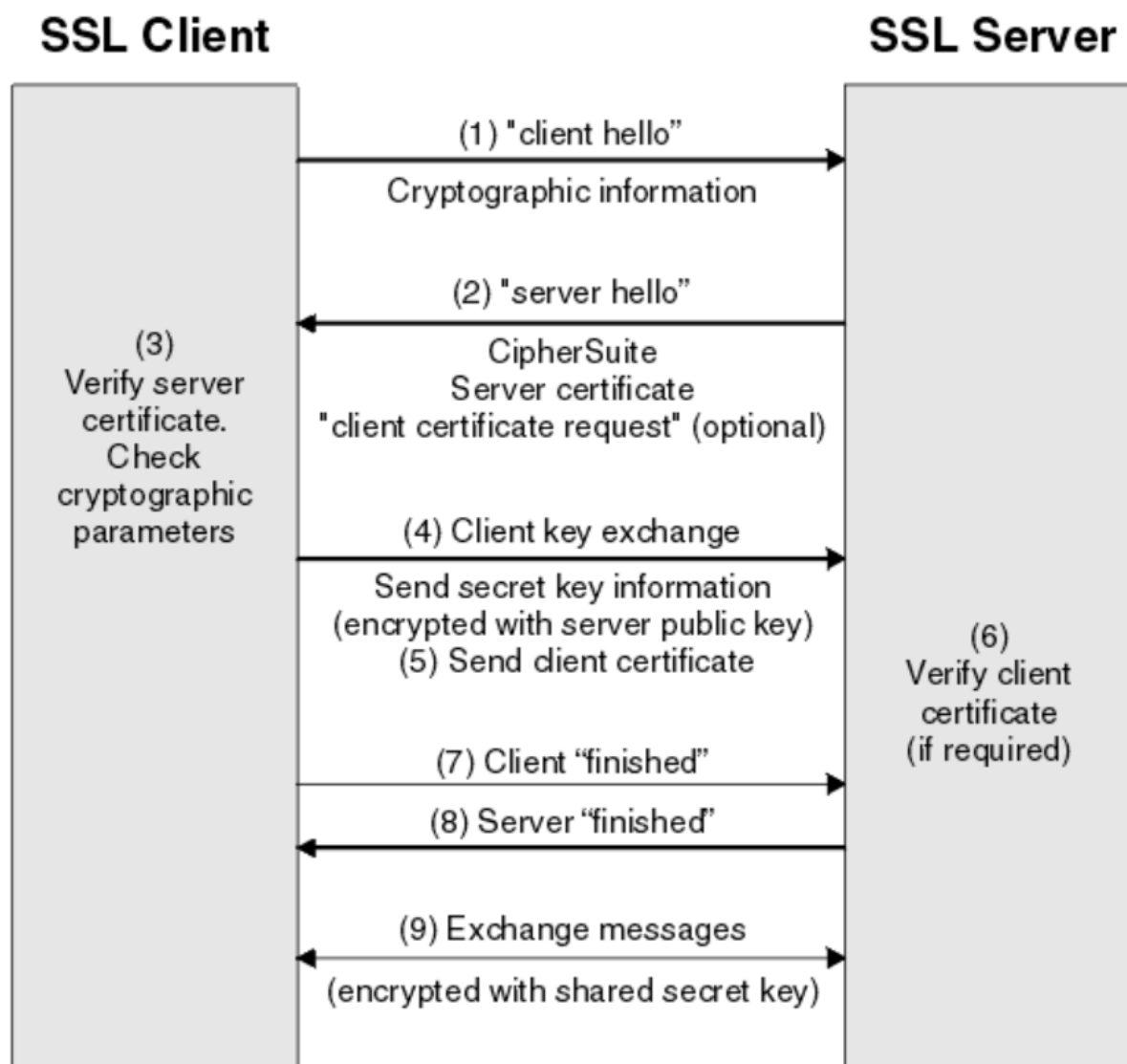
We will first present the TLS 1.2 handshake, then compare it with its successor TLS 1.3 in order to better understand the corrections and improvements made.

The initiation of a communication takes place as follows:

- 1) The client and the server agree on the version of the protocol to be used
- 2) The cryptographic algorithm during future communications is chosen

- 3) The client and the server mutually authenticate in s " exchanging then validating digital certificates.
- 4) Asymmetric encryption is used to generate and share the shared secret key, in order to secure its transmission.
- 5) Once this key has been shared, the client and the server can now communicate by encrypting their messages with the key shared in 4). The encryption is therefore symmetric this time, which is faster than asymmetric encryption.

Here is a detailed diagram showing how a handshake using this protocol takes place:



Handshake up to TLS version 1.2 included

- Step 1: The SSL / TLS client sends a "client hello" message to the server, in which it lists information such as the version SSL / TLS it supports as well as the list of cryptographic suites (set of supported encryption algorithms) in order of preference, from the most robust to the weakest. It also sends the compression methods it

supports. In addition, it sends a string of random bytes ("Client hello") which will make it possible to avoid a "man in the middle" type attack (see step 2)

```

❏ Cipher Suites (34 suites)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
Cipher Suite: TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x0088)
Cipher Suite: TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA (0x0087)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
Cipher Suite: TLS_RSA_WITH_CAMELLIA_256_CBC_SHA (0x0084)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
Cipher Suite: TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x0045)
Cipher Suite: TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA (0x0044)

```

Example of Cypher Suites proposed during a "Client hello"

- Step 2: The server responds with a "hello server" which contains the cryptographic suite it has chosen among those offered by the client, as well as the session identifier (session ID) and a 32-byte random string ("Server Hello"). These random chains are therefore important because they will make it possible to verify that the client is indeed talking to the server and vice versa, as we will explain later.

The server also sends its digitized certificate (X.509), and can request one from the client to authenticate itself by sending a "certificate request", in which it specifies the certificate format it supports. It also sends its public key to the client, which will be necessary for the exchange of keys ("server key exchange"). Finally, it sends a "server Hello Done" indicating that the ServerHello and the associated messages are finished.

- Steps 3, 4 and 5: The client verifies the digitized certificate sent by the server. If the server sent a request for the client's certificate, the client also sends a random octet string encrypted with its private key and containing its digital certificate. (or a "no digital certificate alert"). In some handshakes, the absence of this client authentication will prevent communication from being established. In addition, the client sends another string of random bytes ("pre master secret") which will be used, combined with the "client random" and "server random" (steps 1 and 2), to generate the "master secret" which will be the symmetric key used for future communications. This "pre secret master" string is encrypted with the public key of the server. This whole step is called the "client key exchange".

We then understand the role of the "random client" and "random server" at the beginning. If they had not taken place, all the generation of the final secret key would

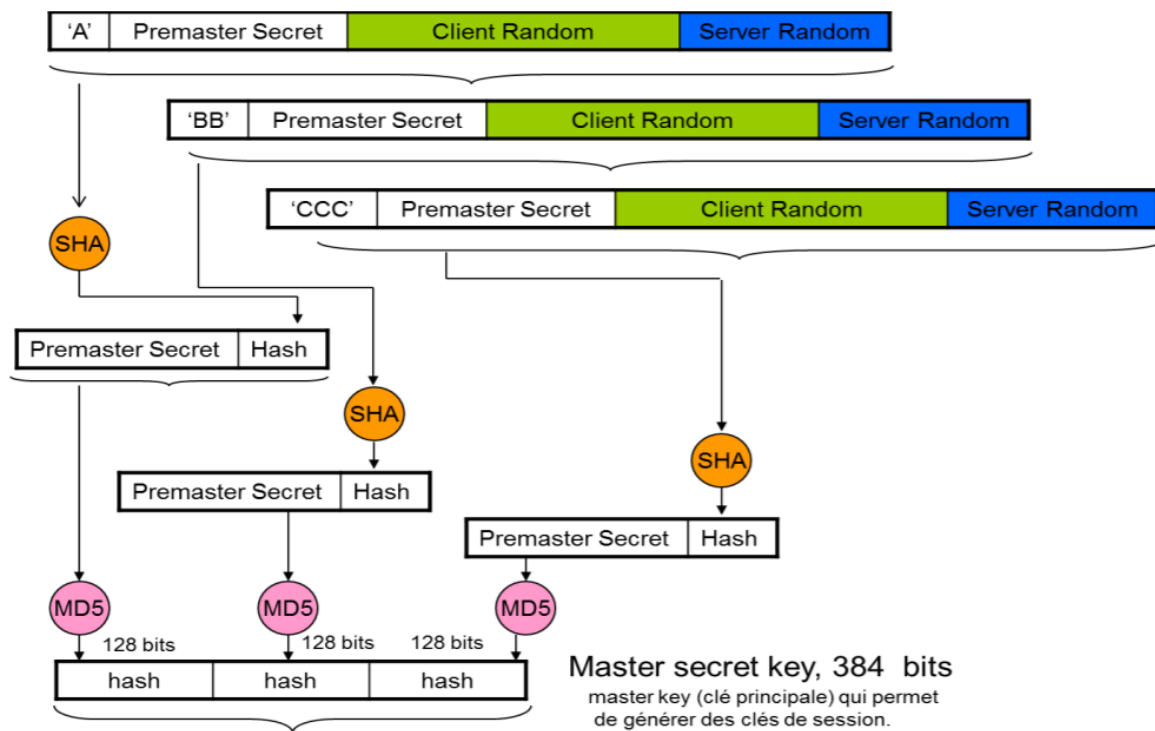
have gone to the client, who sends the “pre master secret”. Thus, if the server itself generated messages with an encryption set by the client, in a “man in the middle” type attack, we could imagine an attacker who collects the “pre master secret” key sent by the client and pretends to be the server without this being detected by the client (“replay” attack). For this, it is important that there is a construction of the key based on both sides, and different for each communication: we speak of “generation of randomness”. If the server sends a random string to the client at the beginning, the latter can verify afterwards that it is indeed the same server which sent the random string at the beginning, since this one is used in the design of the final key “secret master” and that moreover only the server could decode the “pre master secret” with its private key.

- Step 6: The SSL / TLS server verifies the client's certificate if requested. He decodes the “pre master secret” with his private key (he is the only one able to do so. The master secret can then be calculated from both sides:

master_secret = PRF (pre_master_secret, "master secret", ClientHello.random + ServerHello.random)

where PRF is a pseudo-random function

Below is a diagram explaining in detail the calculation of this “Master secret”:



Generation of the Master secret

- Step 7: The client then sends the server a “final” message (“client finished ”), Which is encrypted with the new secret key calculated as described above (“ change cipher spec ”). With this step, the client indicates that the handshake is finished on his side, and can verify that the server was able to decode the message with their common secret key
- Step 8: The server sends the client a “server finished” message which is also encrypted with the new common secret key (“change cipher spec”), and indicates this so that the handshake is also complemented his side9.:
- Step For the remainder of the conversation, the client and the server can now exchange symmetrically encrypted messages with the shared secret key.

During the next connections, new keys can be derived from the “Master Secret” using for example the MD5 hash function combined with new random strings generated on the server side and on the client side. Note that the previous session is stored in cache by the server, and can be found in particular from the Session_ID (which is initialized to 0 during the first connection)

We note that the handshake took two RTT (Round-Trip-Time) to be complete. In general, we estimate this duration between a quarter and a half second, but this duration can vary according to other factors. When we compare the TTFB (Time-To-First-Byte) between HTTP and HTTPS, we see that the establishment of the connection with HTTPS is longer, which is normal.

2) TLS 1.3

a) From TLS 1.2 to TLS 1.3

Two problems with TLS 1.2 led to its evolution: on the one hand, the fact that the initiation of any TLS 1.2 connection lasts 2 round trips, noting that in terms of encryption the first round trip is only used to choose the cryptographic suite without the construction of the shared secret (“master secret”) being really advanced. Furthermore, the property of perfect forward secrecy is not respected with RSA type ciphers or even the Diffie-Hellman key exchange of conventional type. Indeed, in these types of encryption, if the secret key of the server is compromised, all the messages between clients and servers will be able to be decrypted.

This is why TLS 1.3 arrived in force with several strong measures:

- the removal of all cryptographic suites deemed too weak in terms of security. A necessary criterion for a crypto suite to be used in TLS 1.3 is that it checks the ownership of PFS (Perfect Forward Secrecy), which corresponds to the second issue

we raised with TLS 1.2. Thus, even the compromise of a private key of the server at a given moment would not be sufficient to be able to find all the keys used on both sides and thus completely decrypt the communications. RSA encryption is therefore of course no longer supported, as well as static DH. On the other hand, the Diffie-Hellman key exchange known as “ephemeral” is supported: it consists in that the private keys exchanged during a communication are for single use. Thus, the compromise of a key at a given moment will not make it possible to decrypt all previous communications.

TLS 1.3 finally only supports 5 Ciphersuites:

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_AES_128_CCM_SHA256
- TLS_AES_128_CCM_8_SHA256

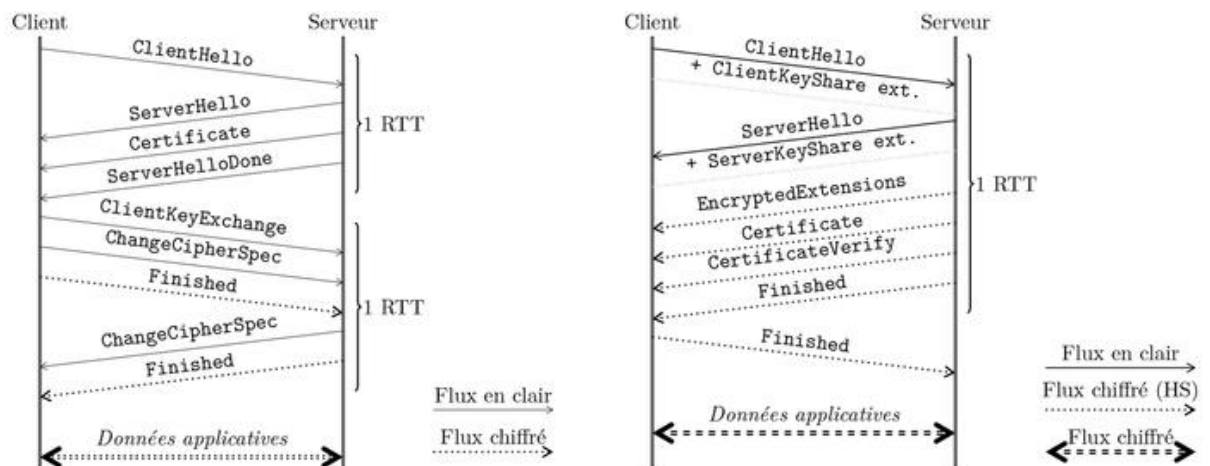
- the reduction in the duration of the handshake, which becomes much more optimized by having only one round trip instead of two for TLS 1.2. The exchange of keys and the definition of the cryptographic primitive used for the exchanges is therefore done from the first handshake.
- In addition, we also note the appearance of the “0-RTT” appearance, which we will focus on later. It allows, between a client and a server that has already communicated before, to send application data from the first round trip.
- It can be mentioned that the compression has also been removed.

We will first of all focus on the new version of the TLS 1.3 handshake, now that we have seen the general principle of a TLS handshake through TLS 1.2.

- The SNI (Server Name Indication) becomes mandatory for TLS 1.3: given that a server can host several domain names, it is appropriate for a server when it receives a request to know which domain name is targeted by the client. This is the role of the SNI, which is an extension present in the Client Hello indicating the domain name of the server with which the client wants to interact.

b) Handshake

To introduce the new handshake, let's start with a diagram showing the difference between the TLS 1.2 and TLS 1.3 handshake.



On the left is the TLS 1.2 handshake, described in the first part. It appears that the pre-communication exchanges last 2 round trips.

On the right, we can observe the TLS 1.3 handshake, which only takes an RTT.

Let's detail the different steps on the right side in order to understand how the TLS 1.3 handshake works.

Step 1: Client Hello + Client Key Share

The client sends his Cipher Suite (set of cryptographic suites offered) as well as a “set” of parameters, made up of Diffie-Hellman groups as well as keys ready to be exchanged using these groups in question. For a given group, it sends several pairs (public key (g, p); f (private key; public key) = $g^a \text{ mod } p$) encrypted by different cryptographic suites (Key Share). These keys are among other things created using a Client.random as in TLS 1.2. All the keys generated are contained in a field called “Key Share Entry”.

During this step, the client also makes an assumption about the cryptographic suite that the server will probably choose. Note that the keys generated here are temporary, we speak of “ephemeral Diffie-Hellman”. This will allow to add even more security compared to a classic Diffie-Hellman, which will allow the principle of Perfect Forward Secrecy (PFS) to be verified.

There is a priori no explicit request to send a certificate at this stage. The use of certificates is optional and will depend on the key exchange method agreed between the client and the server. In practice, in all cases, the client will ask the server to send an X 509 certificate, except when using PSK (Pre Shared Key, see what follows).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.1.117	172.16.1.130	TCP	74	34152 → 4433 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=269767201 TSecr=0 WS=128
2	0.009143	172.16.1.130	172.16.1.117	TCP	74	4433 → 34152 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=269762153 TSecr=269767201
3	0.010254	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=269767212 TSecr=269762153
4	0.010885	172.16.1.117	172.16.1.130	TLSv1.3	301	Client Hello
5	0.012012	172.16.1.130	172.16.1.117	TCP	66	4433 → 34152 [ACK] Seq=1 Ack=236 Win=30080 Len=0 TSval=269762156 TSecr=269767212
6	0.014010	172.16.1.130	172.16.1.117	TLSv1.3	1139	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
7	0.015756	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=236 Ack=1074 Win=31360 Len=0 TSval=269767217 TSecr=269762158
8	0.017415	172.16.1.117	172.16.1.130	TLSv1.3	146	Change Cipher Spec, Application Data
9	0.017879	172.16.1.117	172.16.1.130	TLSv1.3	124	Application Data, Application Data

TLSv1.3 Record Layer: Handshake Protocol: Client Hello Content Type: Handshake (22) Version: TLS 1.0 (0x0301) Length: 230 Handshake Protocol: Client Hello Handshake Type: Client Hello (1) Length: 226 Version: TLS 1.2 (0x0303) Random: 8147c166d51bfa4bb5e02ae1a787131d11aac6cef7fab94... Session ID Length: 32 Session ID: 6f9d07f1953e99d8f36d97ee190b061bf4840bb68fccdee2... Cipher Suites Length: 8 Cipher Suites (4 suites)	0000 b8 27 eb 45 99 91 00 0c 29 dd 61 7a 08 00 45 00 .'.E...).az..E. 0010 01 1f 85 0b 40 00 40 06 59 b6 ac 10 01 75 ac 10@.Y....u.. 0020 01 82 85 68 11 51 7f 2a d7 29 ec 60 a9 af 80 18 ...h.Q.*.).`... 0030 00 e5 2a 04 00 00 01 01 08 0a 10 14 52 2c 10 14 ..*.....R,.. 0040 3e 69 16 03 01 00 e6 01 00 00 e2 03 03 81 47 c1 >i.....G. 0050 66 d5 1b fa 4b b5 e0 2a e1 a7 87 13 1d 11 aa c6 f...K...*..... 0060 ce fc 7f ab 94 c8 62 ad c8 ab 0c dd cb 20 6f 9db.....O.. 0070 07 f1 95 3e 99 d8 f3 6d 97 ee 19 0b 06 1b f4 84 ...>...m..... 0080 0b b6 8f cc de e2 d0 2d 6b 0c 1f 52 53 13 00 08-k...RS.. 0090 13 02 13 03 13 01 00 ff 01 00 00 91 00 00 00 0cdogfish.... 00a0 00 0a 00 00 07 64 6f 67 66 69 73 68 00 0b 04 00b0 03 00 01 02 00 0a 00 0c 00 0a 00 1d 00 17 00 1e
---	--

Capture Client Hello TLS 1.3 (Cloudshark)

Among other things, we notice the presence of the Session_ID and a list of 4 proposed Cipher Suites (among the 5 in total accepted in TLS 1.3).

We can make another remark: the fact that the session identifier Session_ID is non-zero during the Client Hello comes from the fact that our capture is carried out using the Cloud Shark online tool, which behaves like a “Middlebox”(network that inspects, filters, handles packets). The TLS 1.3 RFC (<https://tools.ietf.org/html/rfc8446#appendix-D.4>) explains that for Middleboxes, certain particular behaviors are observed, such as the need to send a non-zero session identifier even during Client Hello. For a classic handshake, this session identifier is zero during Client Hello, and it is the server that creates it during Server Hello.

▼ Cipher Suites (4 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_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)

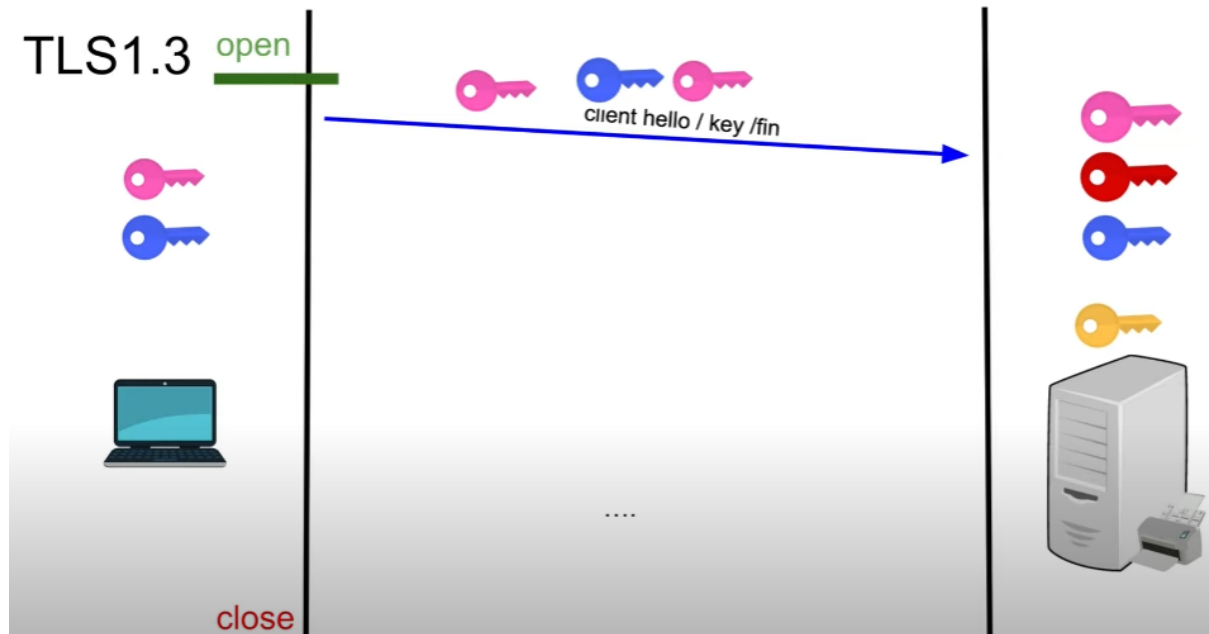
There is also a list of possible extensions, including Key Share:

```
Extensions Length: 145
▶ Extension: server_name (len=12)
▶ Extension: ec_point_formats (len=4)
▶ Extension: supported_groups (len=12)
▶ Extension: SessionTicket TLS (len=0)
▶ Extension: encrypt_then_mac (len=0)
▶ Extension: extended_master_secret (len=0)
▶ Extension: signature_algorithms (len=30)
▶ Extension: supported_versions (len=7)
▶ Extension: psk_key_exchange_modes (len=2)
▶ Extension: key_share (len=38)
```

Step 2: Server Hello + Server Key Share (+ optional Certificate + optional Certificate Request + optional Certificate Verify + EncryptedExtensions + Hello Retry in case of failure) + Server Finished

The server receives the shared key from the client. It can then build its own key (in this case key pair with Diffie-Hellman) with the cryptographic primitive that it has chosen among those offered by the client (if it supports at least one). This key (Key Share) must be created using the same cryptographic suite as the client, and if DH “ephemeral” was used by the client, the server must also use the same Diffie-Hellman group to share f (private key server = b ; public key = (g, p)) = $g^b \bmod p$. If ever none of the primitives proposed by the client is supported by the server, the latter sends a “Hello Retry” message so that the client makes a new Client Hello with another cryptographic primitive that it supports. If the client and the server do not share any cryptographic primitive (rare case), the server sends an Alert message and the communication is stopped.

Note that at this time, the server then has the two Key Share (client and server), it can therefore build the final symmetric key, the master secret.

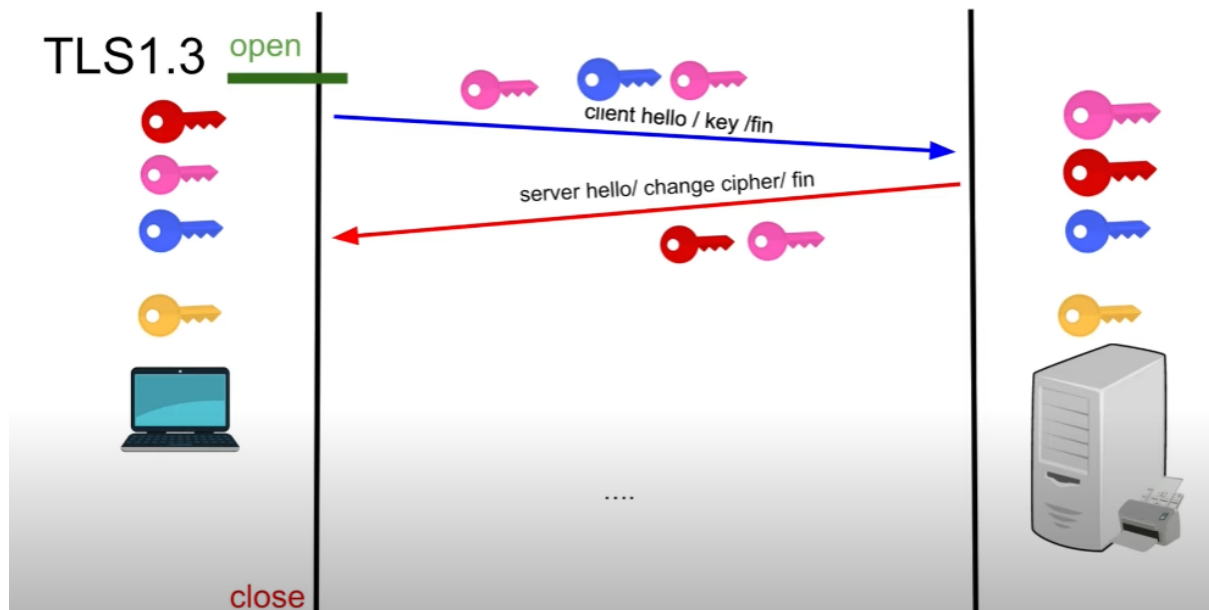


-  Clé privée serveur
-  Clé publique client
-  Clé privée client
-  Clé symétrique finale

In this diagram, we can observe the mixed pair of public / private keys of the client. The pink key is the public key, the blue key is his private key. What it sends in clear is the pair made up of its pink public key, as well as the public / private key mixture (pink key + blue key). From this pair, it is impossible to trace the blue private key (this would amount to solving a mathematically complex Diffie-Hellman problem) for an attacker.

The server, for its part, generates a temporary private key, in red in this diagram. By combining it with the blue key / pink key pair sent by the client, he can build the final symmetric key (or at least which will be given as input to the cryptographic suite chosen to build the final key)! It is the yellow key in the diagram.

The server then sends the client the mix of red key (its private key) / client's public key. Once again, from this mix of keys, the private key of the server cannot be found. Note also when in mode Diffie-Hellman, the server proves that it is indeed the owner of its private key by using a digital signature allowing in addition to verify the integrity of the exchange: this prevents the known man in attack. the middle on Diffie-Hellman. He signs the entire handshake with his private key in the Verify certificate. Here then is the situation:



By combining the red key / pink key pair sent by the server as well as its blue private key, the client can build a shared key. It is the same as the one created by the server! This is due to the associativity of the operations brought into play by Diffie-Hellman. (as well as cryptographic suites). To obtain the final symmetric key which will be used for the encryption of the messages, on each side, we mix the shared secret (PMS) with the transcript-hash of the key exchange, this gives an HMS. From the HMS and the randoms sent, we produce a `tk_hs` which will be the final symmetric key. This context changing each time through the `ClientHello.random` and `ServerHello.random` ensures that the final shared symmetric key is very different for each handshake.

Thus, we observe that the shared symmetric key is indeed created at client and server level, and this in a single round trip! The use of Diffie-Hellman ("ephemeral") therefore has a double contribution: a gain in terms of security (PFS) but also in terms of number of exchanges, since the construction of the final symmetric key was carried out in one only round trip! This is the great contribution of TLS 1.3.

The server must also send the Encrypted Extensions, which constitute the first message that will be encrypted using keys derived from the PMS. The customer should absolutely check the Encrypted Extensions to ensure that none of the extensions used are prohibited, in which case an Alert message "illegal_parameter" will be issued. The Encrypted Extensions can be for example the Server Name, the Message Length ...

During this step, the server can send its certificate ("Certificate") if the chosen exchange method requires it. Note that this certificate is already encrypted since the final symmetric key is constructed by the server at this level. Upon receipt of the Server Hello, the client will also be able to construct this symmetric key and therefore decode the server certificate. he himself also asks the client for an authentication certificate if he considers it necessary ("Certificate Request").

The server also sends a Certificate Verify which makes it possible to prove the possession of the private key. This certificate verify is required if the server sent a certificate. This field also makes it possible to ensure the integrity of the handshake up to this point. Indeed, the server signs the transcript-hash with its private key, which is the hash of the concatenated handshake messages.

The server then sends a “Server Finished”. This finished is a MAC on the whole handshake. It allows to show the good possession of the PMS $g^a(b)$. Indeed, this Server finished is calculated as follows: ServerFinished = HMAC (finished_key,

Transcript-Hash (Handshake Context,
Certificate *, CertificateVerify *)

finished_key is derived from PMS $(g^a(b))$. This makes it possible to link the identities of the 2 points.

No.	Time	Source	Destination	Protocol	Length	Info
2	0.009143	172.16.1.130	172.16.1.117	TCP	74	4433 → 34152 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=269762153 TSecr=269762153
3	0.010254	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=269762153 TSecr=269762153
4	0.010885	172.16.1.117	172.16.1.130	TLShv1.3	301	Client Hello
5	0.012012	172.16.1.130	172.16.1.117	TCP	66	4433 → 34152 [ACK] Seq=1 Ack=236 Win=30080 Len=0 TSval=269762156 TSecr=269762122
6	0.014010	172.16.1.130	172.16.1.117	TLShv1.3	1139	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
7	0.015756	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=236 Ack=1074 Win=31360 Len=0 TSval=269762158 TSecr=269762158
8	0.017415	172.16.1.117	172.16.1.130	TLShv1.3	146	Change Cipher Spec, Application Data
9	0.017879	172.16.1.117	172.16.1.130	TLShv1.3	124	Application Data, Application Data
10	0.024132	172.16.1.130	172.16.1.117	TLShv1.3	331	Application Data

Handshake Protocol: Server Hello Handshake Type: Server Hello (2) Length: 118 Version: TLS 1.2 (0x0303) Random: 3964dbec5022bfb0783a15f8fd02518c8cf05be901c389b... Session ID Length: 32 Session ID: 6f9d07f1953e99d8f36d97ee190b061bf4840bb68fccdee2... Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302) Compression Method: null (0) Extensions Length: 46 Extension: supported_versions (len=2) Extension: key_share (len=36)	0000 b8 27 eb 45 99 91 00 0c 29 ee 6c 65 08 00 45 00 .'.E....).le..E. 0010 04 65 97 c5 40 00 40 06 43 b6 ac 10 01 82 ac 10 .e..@.@.C..... 0020 01 75 11 51 85 68 ec 60 a9 af 7f 2a d8 14 00 18 ..u.Q.h.'...'*. 0030 00 eb 48 48 00 00 01 01 08 0a 10 14 3e 6e 10 14 ..HH.....>n.. 0040 52 2c 16 03 03 00 7a 02 00 00 76 03 03 39 64 db R.....Z...V..9d. 0050 ec 50 22 bf bd 07 83 a1 5f 8f d0 25 18 c8 cf 05 .P".....%. 0060 be 90 1c 38 9b 8a 28 46 39 e3 7c db 66 20 6f 9d ...8..(F9. .f o. 0070 07 f1 95 3e 99 d8 f3 6d 97 ee 19 0b 06 1b f4 84 ...>...m..... 0080 0b b6 8f cc de e2 d0 2d 6b 0c 1f 52 53 13 13 02-k..RS.... 0090 00 00 2e 00 2b 00 02 7f 1c 00 33 00 24 00 1d 00+.---3.\$.... 00a0 20 ee d2 11 97 9a c7 94 1f dd de 11 1c d4 f5 b9 00b0 81 ec d0 69 bb f9 e3 ef f2 b4 2d 01 cb 6b 9e 57 ...i.....k..W
--	--

Capture Server Hello (Cloudshark)

We can in particular observe the Cipher Suite selected by the server (TLS_AES_256_GCM_SHA384) which is indeed part of the 4 Cipher Suites offered by the client in the Client Hello.

We also notice a session identifier Session_ID identical to that of the Client Hello, which confirms that the 2 indeed have a common session.

Step 3: Client Finished (+ optional Certificate + optional Certificate Verify) + Application Data

At this level, if the server accepted one of the proposed cryptographic primitives, the client was able to construct the shared symmetric key using the process explained below. above. It can with this key decode the server certificate (if needed). The client can also send his own certificate, encrypted with the shared secret key (PMS master secret) if the server has made a Certificate Request.

If the client sends a certificate, it must also send a Certificate Verify like that of the server: a signature by its private key of the context.

The client finally sends a Finished Client in the image of the Finished Server.

From this moment, the transfer of application data can begin.

Note that if the server has not retained any of the primitives proposed by the client, and therefore sent a Hello Retry, this step will be a new Client Hello with a new proposal for CipherSuite by the client. If no agreement is found even after this new Client Hello, communication is stopped following an Alert message from the server.

No.	Time	Source	Destination	Protocol	Length	Info
5	0.012012	172.16.1.130	172.16.1.117	TCP	66	4433 → 34152 [ACK] Seq=1 Ack=236 Win=30080 Len=0 TSval=269762156 TSecr=269767212
6	0.014010	172.16.1.130	172.16.1.117	TLSv1.3	1139	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application
7	0.015756	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=236 Ack=1074 Win=31360 Len=0 TSval=269767217 TSecr=269762158
8	0.017415	172.16.1.117	172.16.1.130	TLSv1.3	146	Change Cipher Spec, Application Data
9	0.017879	172.16.1.117	172.16.1.130	TLSv1.3	124	Application Data, Application Data
10	0.024132	172.16.1.130	172.16.1.117	TLSv1.3	321	Application Data
11	0.024250	172.16.1.130	172.16.1.117	TLSv1.3	321	Application Data
12	0.024500	172.16.1.130	172.16.1.117	TCP	66	4433 → 34152 [FIN, ACK] Seq=1584 Ack=375 Win=30080 Len=0 TSval=269762168 TSecr=269767219
13	0.025876	172.16.1.117	172.16.1.130	TCP	66	34152 → 4433 [ACK] Seq=375 Ack=1585 Win=35712 Len=0 TSval=269767227 TSecr=269762168

Transmission Control Protocol, Src Port: 34152, Dst Port: 4433, Seq: 236, Ack: 1074

Secure Sockets Layer

▼ TLSv1.3 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec

Content Type: Change Cipher Spec (20)

Version: TLS 1.2 (0x0303)

Length: 1

Change Cipher Spec Message

▼ TLSv1.3 Record Layer: Application Data Protocol: Application Data

Opaque Type: Application Data (23)

Version: TLS 1.2 (0x0303)

Length: 69

Encrypted Application Data: 3e52f513cc90c0b7064d43c6543b53a547397d8b93ce

0000 b8 27 eb 45 99 91 00 0c 29 dd 61 7a 08 00 45 00E....).az..E..

0010 00 24 85 0d 40 00 40 06 5a 4f ac 10 01 75 ac 10@.@.ZO...U...

0020 01 82 85 68 11 51 7f 2a d8 14 ec 60 ad e0 80 18 ...h.Q*.....

0030 00 f5 73 d6 00 00 01 01 08 0a 10 14 52 33 10 14 ...S.....R3...

0040 3e 6e 14 03 03 00 01 01 17 03 03 00 45 3e 52 f5 >n.....E)R..

0050 13 cc 90 0c 0b 70 64 d4 3c 65 43 b5 3a 54 73 97pd.<eC.:Ts.

0060 d8 b9 3c e9 fb 72 3a ca 83 e1 dd a1 ba 45 f4 be ...r.r.....E...

0070 1f a6 94 77 6a 08 21 1c c9 5b 11 fd f7 b1 fc a5 ...w3.l.l.....

0080 f4 05 47 df f5 3f de 0f 22 db 57 21 b8 f9 73 be ..G..?..".Wl...s.

0090 3f db ?.

Capture Change Cipher Spec (TLS 1.2 + Application Data) (Cloudshare)

This capture shows the second RTT during an exchange with TLS 1.3. We notice the presence of Application Data from the second round trip, which is characteristic of TLS 1.3. Stranger fact: we notice the existence of a Change Cipher Spec field, which is obsolete since TLS 1.3. This is again due to the remark made above, concerning the fact that “Middleboxes” adopt a particular behavior: to quote the RFC “ Implementations can increase the chance of making connections through those middleboxes by making the TLS 1.3 handshake look more like a TLS 1.2 handshake. ” It is for this reason that we observe the Change Cipher Spec field, which is in fact a “dummy Change Cipher Spec”, that is to say a CCS “camouflaged” to give the impression to the server of be in TLS 1.2 and maximize the chances of establishing a connection with the client.

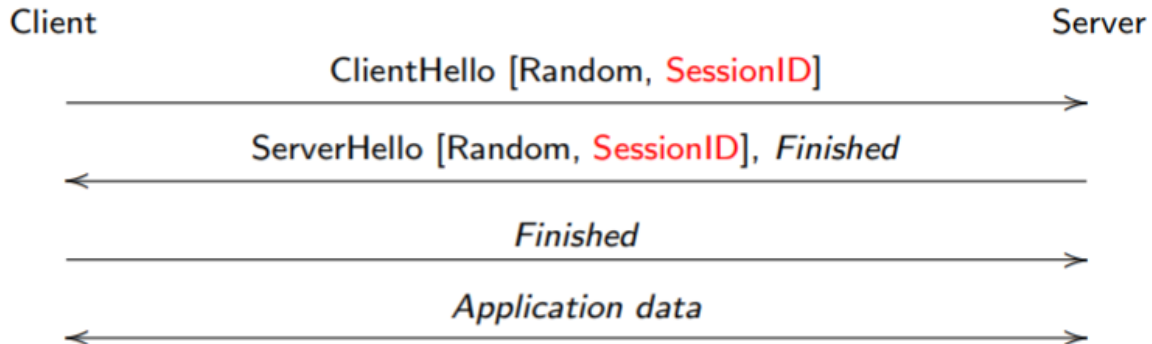
c) Resuming aconnection

TLS 1.2

For TLS 1.2, connection recovery was managed in 2 possible ways:

- either by using the session identifier Session_ID, kept in the server cache. The recovery exchange lasted 1 RTT, during which the client sent a new Client.random as well as the Session_ID, while the server checked in its cache that they did indeed have a common session with the Session_ID in question, while sending also in turn a new Server.random. This made it possible to build a new shared symmetric key, derived from these two Randoms as well as from the old master secret. This possibility presented a significant risk insofar as the compromise of the master secret

or the private key of the server in the meantime would allow an attacker to gain access to the new master secret. Note that it is recommended that the session identifier remain in the server cache for 24 hours, otherwise the client and the server will have to perform a full handshake again.



- Another possibility, easier to implement in terms of memory for the server, is the use of a session ticket. Indeed, in the implementation with the Session_ID, the server is responsible for storing the Session_IDs of all the clients with which it has communicated for a given duration. This poses scalability problems for the server since the Internet has a very large number of users per second. The idea of using a ticket is that it is the client who will have to keep in mind what to be able to connect to the server again. At the end of a TLS handshake, the server sends a ticket to the client along with key information about the current session, encrypted with its private key. The client will keep the ticket and associated information in cache memory, and when it wants to resume a session with the server, it will send it this information that only the server can decode. The session can then resume

TLS 1.3:

Under TLS 1.3, you should know that the resumption of session via session tickets or a session identifier is obsolete. These two methods are replaced by what is called a PSK (Pre-Shared-Key) mode. This PSK already existed in TLS 1.2 but was not used in this context, only for certain IoT type applications.

At the end of the handshake of a previous session, the PSK is exchanged between the client and the server in order to be used by the client during a new connection. This PSK is built by the server from a key derived from the master secret, then sent to the client, which keeps it in cache. It also contains information about configuring the server. The client can then subsequently use this PSK identifier which, if accepted by the server, will allow a secure connection based on cryptographic parameters used during the last session. The key derived from the master secret will be used to encrypt new communications.

The client sends one or more PSK identities as an opaque data set. These can be database lookup keys (similar to session identifiers) or self-encrypted and self-authenticated values (similar to session tickets). If the server accepts one of the given PSK identities, it responds

with the one it has selected. The KeyShare extension is sent to allow servers to ignore PSKs and revert to a full handshake.

In addition, the PSK can be combined with an exchange key of the Diffie-Hellman type, which makes it possible to verify the property of Perfect Forward Secrecy, which is not guaranteed when using the PSK alone.

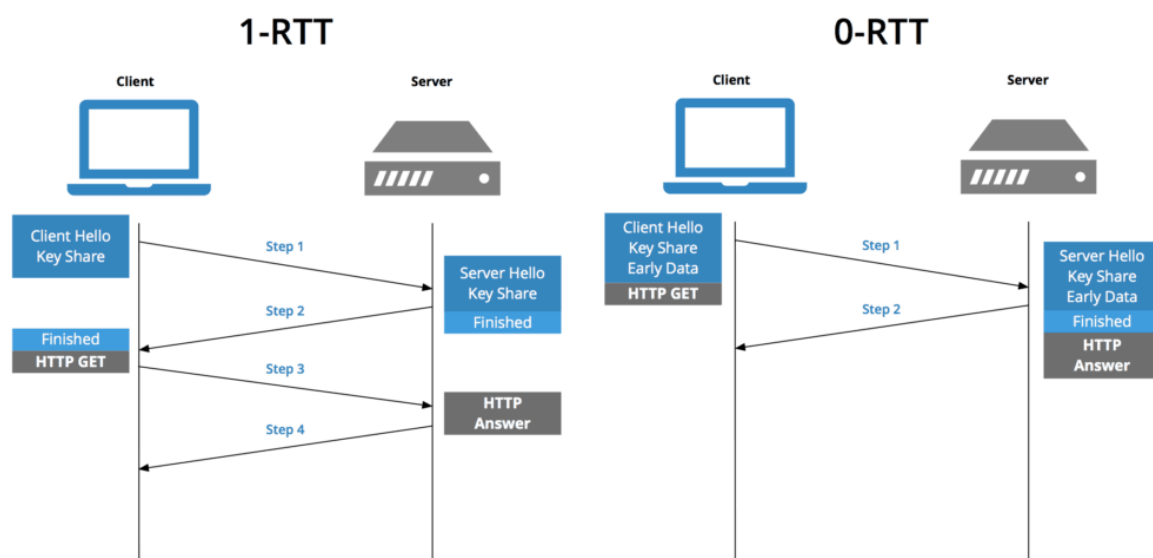
The handshake takes place according to the classic session initiation scheme, except that:

- The client sends a Pre Shared Key identity specifying the sending mode (PSK only or PSK + keyshare)
- The server sends the Pre Shared Key Identity chosen
- No certificate is involved (neither Certificate, Certificate Request, Certificate Verify)

d) “0-RTT” option

The 0-RTT is one of the flagship options of TLS 1.3. It allows a client and a server to send application data to each other on the first round trip when resuming a session, if they have already communicated before. This is an option inspired by the QUIC protocol offered by Google. It is based on the use of the PSK presented in part b). The connection recovery will take place in 0-RTT mode if, during the Client Hello, the client sends the extension “early_data”. In this case, it can directly send application data, to which the server will respond successfully if its PSK is accepted.

This represents a gain of a round trip compared to the classic recovery mode, also present in TLS 1.2.



However, we can notice that the 0-RTT is below the other measures presented by TLS 1.3 in terms of security. Indeed:

- the PFS is not verified, given that the first exchanged application data is encrypted with the derived key contained in the PSK and provided by the server during the previous connection. As this is based on the private key of the server, a compromise of this one would allow an attacker to decrypt the sent messages.
- there is no guarantee of protection against a “replay” attack. Indeed, under the classic session resumption of type 1-RTT, Server random made it possible to serve as a nonce preventing replay.

1) Management of cryptographic services

a) Management of authentication

TLS 1.2:

In the case of server authentication: the client uses the public key that the server gave him during the “server hello” in order to encrypt the data which will be (in particular the pre-master secret) which will be used to generate the secret key. Thus, the server can generate the secret key with the client if and only if it can decode this data with its private key. (which he is the only one to have).

In the case of client authentication: the server uses the public key provided in the client's certificate in order to decode the data that the latter sent to it during step 5. The principle is the same, the authentication is then assured.

Exchanging “finished” messages with the shared key ensures that the authentication is complete. In addition, the authentication is complete if and only if each of the steps of the handshake has been carried out correctly.

The role of certificates also makes it possible to guarantee authentication even more. Their management is quite complex, but the bottom line is that certificates SSL play a very similar role to client certificates. However, in the case of a client, the certificate is used to identify the client / individual itself while in the case of a server the certificate authenticates the owner of the site.

If we call CA Client the certification authority of the Client and CA Server the certification authority of the Server, in the case of client and server authentication, the server needs:

- a personal certificate provided by CA Server
- from its private key
- of a client certificate provided by CA Client

The client needs:

- a personal certificate provided by CA Client
- his private key
- a server certificate provided by CA Server

It can be seen that in terms of authentication, exchanges are made each time in a similar way for the client and the server.

The certificates contain various information such as a serial number, a validity date, personal data on the owner (name, address, e-mail...)

TLS 1.3:

The server must always authenticate itself, for the client this is optional. Authentication can take place in the case of the TLS 1.3 handshake using asymmetric cryptography.

The server can authenticate via an x509 certificate in the session initialization.

The Certificate Verify makes it possible to ensure that the server possesses the private key that it claims to have.

In the case of a session resumption, the client sends the PSK identity as well as a PSK binder (hashed of the context and of a key derived from the PSK) allowing to prove to the server that it does have the PSK.

The server, for its part, also makes it possible to prove to the client that it does have the PSK by means of the Server Finished which is always mandatory and which is part of the authentication block. Indeed, in the case of a resumption of session, the Server Finished is calculated by $H(\text{base_key}, \text{transcript-hash})$. The base_key is derived from the PSK and the context. So the client can check the hash to make sure the server has the PSK.

b) Certificate verification

There are 4 stages during certificate verifications:

- verification of the digital signature
- the "certificate chain" is verified: normally the client and the server must have had intermediate certificates
- on the expiration date, activation and the validity period are checked
- the certification revocation status

c) Confidentiality management

On the one hand, we observe that the exchange is based on an alternation of symmetric and asymmetric ciphers, which reinforces the security of the communication.

In order to minimize the risk in the event of theft or alteration of the secret key, the latter can be renegotiated periodically, and the old one is then no longer valid (Randoms at regular intervals for TLS 1.2 and 1.3 + "ephemeral Diffie- Hellman "for TLS 1.3). The same

principle of security is used when sending random strings as explained above, which make it possible to avoid “man in the middle” attacks.

During the handshake, the client and the server agree on the cryptographic primitive used as well as the secret key which will therefore be used only for the current session. All the messages during the session will therefore be encrypted with both the chosen primitive and the shared secret key, which ensures that the message remains private and will not be intercepted. Random chains only further decrease the already low risk of interception.

Furthermore, for TLS 1.2, the fact that the encryption used to transmit the private key is asymmetric avoids any key distribution problem (no risk of interception).

For TLS 1.3, the use of Diffie-Hellman guarantees confidentiality, and even Perfect Forward Secrecy, insofar as the keys transferred in the clear (public key pair / public key + private key mix) do not make it possible to go back to the private key for an attacker, as they would require solving a computationally complex mathematical problem.

d) Integrity management

Integrity is mainly enabled by the hash calculation of messages with secret key (hmac). The hash functions used are provided in the list of cryptographic suites used by the client, and it is then the server which will make the final choice, which it will communicate to the client. The hash function used is therefore clearly defined on both sides. For this, however, it is important that the list of cryptographic algorithms supported by the client is not of type “None”, otherwise the integrity is strongly compromised.

In TLS 1.3, the integrity of the handshake is also ensured by the HMACs of the transcript-hash, whether in the certificate verify, or in the Finished.

e) Management of non-replay in TLS 1.3

For 1-RTT, non-replay is provided by the sequence numbers contained in the packets during the same connection.

For 0-RTT however, it is possible to perform a “replay” attack.

II Benchmarking

1) Introduction

The aim of this part is to measure the performance of TLS 1.3 at medium scale. For this, we need to set up a virtual simulation environment. We use 2 Kali virtual machines both in bridge mode on a private LAN. The first Kali machine acts as client (s) and the second as

server. The goal is to simulate HTTP / 2 traffic based on TLS1.3 with several clients and several requests per client.

To measure performance, we can look at several parameters: the connection time, the number of requests processed per second, the response time to a request, and the TTFB (Time to first byte), which corresponds to the time between the moment where the client sends its HTTP request and where it receives the first byte of the server response.

This TTFB is made up of the connection time, the time for sending the HTTP request, and the time for receiving the first response byte. We try to have this TTFB as low as possible because it is an indicator of a well configured server application.

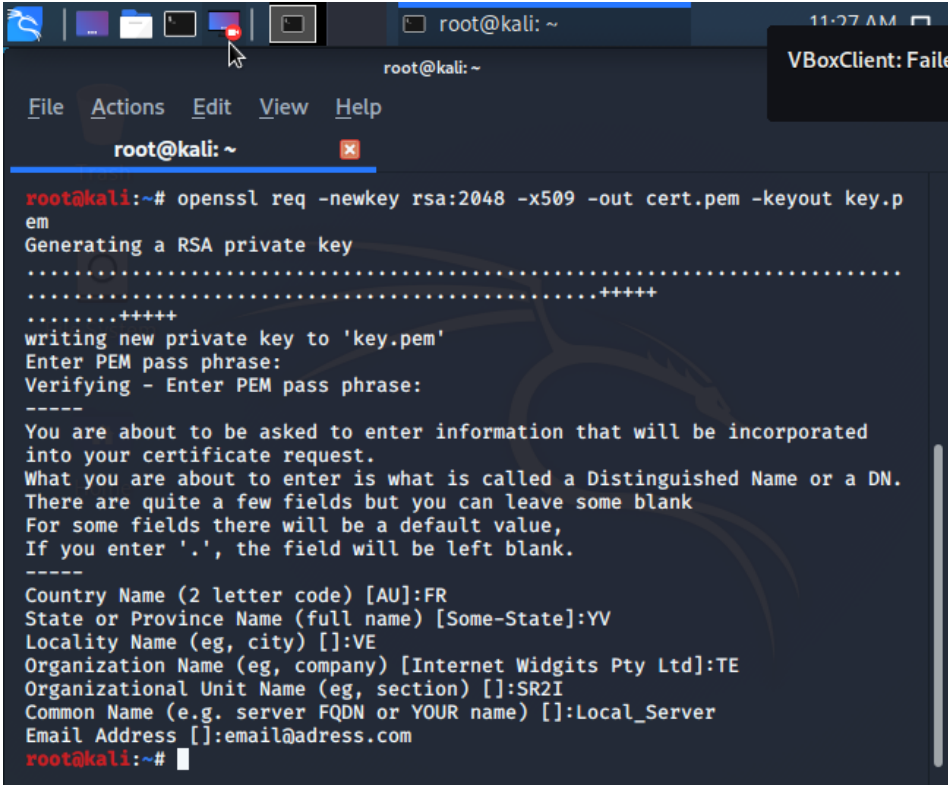
If we want to focus on the performance of TLS 1.3 only, we can focus only on the connection time and the performance of the encryption algorithms used.

2) Connection time:

Let's start by studying the connection time in the simple case of a single client.

First, we need to set up an HTTP server that supports TLS 1.2 and TLS 1.3. For this we use openssl.

First, we need to generate a self-signed certificate and a key pair in pem formats. For that, we use the command:



```
root@kali: ~
File Actions Edit View Help
root@kali: ~
root@kali:~# openssl req -newkey rsa:2048 -x509 -out cert.pem -keyout key.p
em
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:YV
Locality Name (eg, city) []:VE
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TE
Organizational Unit Name (eg, section) []:SR2I
Common Name (e.g. server FQDN or YOUR name) []:Local_Server
Email Address []:email@adress.com
root@kali:~#
```

This done, we can start the https server on port 4433 with the command:

```
root@kali: ~
File Actions Edit View Help
root@kali: ~

.....+++++
.....+++++
writing new private key to 'key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:YV
Locality Name (eg, city) []:VE
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TE
Organizational Unit Name (eg, section) []:SR2I
Common Name (e.g. server FQDN or YOUR name) []:Local_Server
Email Address []:email@address.com
root@kali:~# openssl s_server -accept 4433 -cert cert.pem -key key.pem -www
Enter pass phrase for key.pem:
Using default temp DH parameters
ACCEPT
█
```

First we want to determine the value of the RTT between the client and the server. For that, we write the following script:

```
total=0
for i in $(seq 10); do
  milliseconds=$(( (time nc 192.168.0.30 4433 ) 2>&1 | sed -n 's/^real.*0m0.0*([0]*)s$/\1/p' ))
  (( total += milliseconds ))
done
echo $((total / 10))
```

This script allows using the time command to measure the RTT between the client and the server thanks to an average over 10 tests.

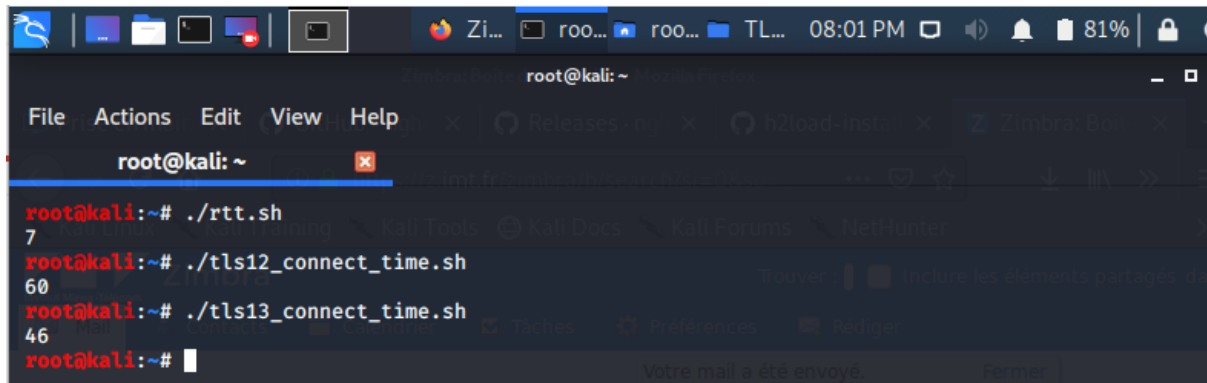
This done, we write similar scripts by replacing the time nc command by the command:

```
time openssl s_client -tls1_2 -connect {Server Address}: 4433
```

or else

```
time openssl s_client -tls1_3 -connect {Server Address}: 4433
```

The results are:



```
root@kali: ~  
File Actions Edit View Help  
root@kali: ~  
root@kali:~# ./rtt.sh  
7  
root@kali:~# ./tls12_connect_time.sh  
60  
root@kali:~# ./tls13_connect_time.sh  
46  
root@kali:~#
```

We see that the connection via TLS 1.3 takes almost 14 ms less than TLS 1.2. In theory, the difference should only be an RTT is 7ms. However, you also have to take into account the time it takes for the server (and client!) Side process to process each request. The more packets the server receives from the client side, the more processing it has to do. It is also necessary to count the noise of the measurement as well as the network congestion. There is therefore a time of 60ms connection (TLS1.2) = 46ms (TLS1.3) + 7ms (1RTT) + 7 ms (processing time of 2 additional queries congestion + + bruit_mesure)

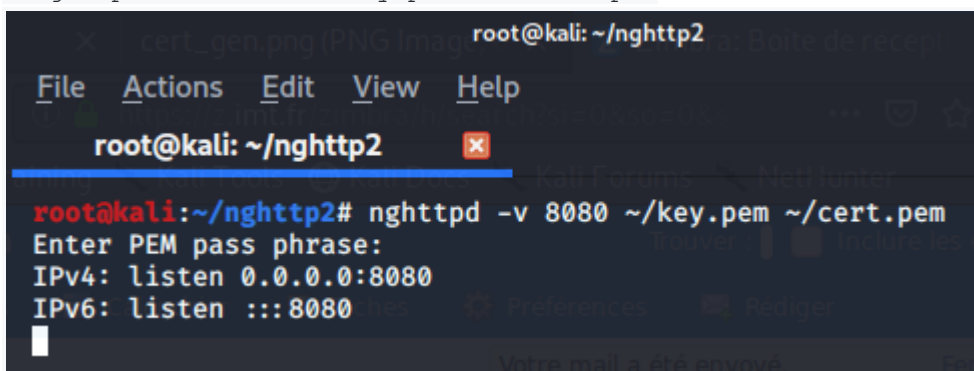
3) Modeling of Clients

Multipletime measurement connection for a client is of course important. However, the protocol is meant to be deployed on a large scale. It is therefore necessary to measure the performance of this protocol in the context of several clients each making several requests on the same server.

The tool we'll be using is h2load (<https://github.com/nghttp2/nghttp2>). As the openssl server opened earlier does not support HTTP / 2, we decide to use the same module, nghttp2, for compatibility reasons.

Thus, we start a server using the command:

```
$ nghttpd -v 8080 ~ / key.pem ~ / cert.pem
```



```
root@kali: ~/nghttp2  
File Actions Edit View Help  
root@kali: ~/nghttp2  
root@kali:~/nghttp2# nghttpd -v 8080 ~/key.pem ~/cert.pem  
Enter PEM pass phrase:  
IPv4: listen 0.0.0.0:8080  
IPv6: listen :::8080
```

On the client side, we can launch the benchmarking with the following command:

```
root@kali: ~/Downloads/nghttp2/src
File Actions Edit View Help
root@kali: ~/...s/nghttp2/src
root@kali:~/Downloads/nghttp2/src# h2load -n1000 -c150 -m50 https://192.168.43.90:8080
```

The -n option allows you to specify the number of requests, the -c option the number of clients, and -m the maximum number of streams per client.

The results are:

```
root@kali: ~/Downloads/nghttp2/src
File Actions Edit View Help
root@kali: ~/...s/nghttp2/src
root@kali:~/Downloads/nghttp2/src# h2load -n1000 -c150 -m50 https://192.168.43.90:8080
starting benchmark ...
spawning thread #0: 150 total client(s). 1000 total requests
TLS Protocol: TLSv1.3
Cipher: TLS_AES_256_GCM_SHA384
Server Temp Key: ECDH P-256 256 bits
Application protocol: h2
progress: 10% done
progress: 20% done
progress: 30% done
progress: 40% done
progress: 50% done
progress: 60% done
progress: 70% done
progress: 80% done
progress: 90% done
progress: 100% done
finished in 1.57s, 637.59 req/s, 118.86KB/s
requests: 1000 total, 1000 started, 1000 done, 0 succeeded, 1000 failed, 0 errored, 0 timeout
status codes: 0 2xx, 0 3xx, 1000 4xx, 0 5xx
traffic: 186.43KB (190900) total, 17.87KB (18300) headers (space savings 85.70%), 147.46KB (151000) data

```

	min	max	mean	sd	12 ju	+/- sd
time for request:	265.49ms	1.13s	643.27ms	195.11ms	12 ju	69.40%
time for connect:	292.43ms	774.43ms	499.25ms	125.55ms	12 ju	62.67%
time to 1st byte:	560.68ms	1.56s	1.14s	285.78ms	12 ju	54.00%
req/s	3.92	12.48	6.28	1.96	12 ju	67.33%

```
root@kali:~/Downloads/nghttp2/src#
```



```
root@kali: ~/Downloads/nghttp2/src
File Actions Edit View Help
root@kali: ~/...s/nghttp2/src x

root@kali:~/Downloads/nghttp2/src# h2load -n2000 -c500 -m100 https://192.168.43.90:8080
starting benchmark...
spawning thread #0: 500 total client(s). 2000 total requests
TLS Protocol: TLSv1.3
Cipher: TLS_AES_256_GCM_SHA384
Server Temp Key: ECDH P-256 256 bits
Application protocol: h2
progress: 10% done
progress: 20% done
progress: 30% done
progress: 40% done
progress: 50% done
progress: 60% done
progress: 70% done
progress: 80% done
progress: 90% done
progress: 100% done

finished in 8.04s, 248.66 req/s, 48.44KB/s
requests: 2000 total, 2000 started, 2000 done, 0 succeeded, 2000 failed, 0 errored, 0 timeout
status codes: 0 2xx, 0 3xx, 2000 4xx, 0 5xx
traffic: 389.65KB (399000) total, 47.85KB (49000) headers (space savings 80.86%), 294.92KB (302000) data

min max mean sd +/- sd
time for request: 973.43ms 4.83s 2.40s 376.23ms 79.20%
time for connect: 1.29s 6.88s 2.74s 792.95ms 79.40%
time to 1st byte: 3.49s 8.03s 5.14s 933.95ms 75.40%
req/s : 0.50 1.15 0.81 0.15 76.00%
```

```
root@kali: ~/Downloads/nghttp2/src
File Actions Edit View Help
root@kali: ~/...s/nghttp2/src x

root@kali:~/Downloads/nghttp2/src# h2load -n10000 -c1000 -m100 https://192.168.43.90:8080
starting benchmark...
spawning thread #0: 1000 total client(s). 10000 total requests
TLS Protocol: TLSv1.3
Cipher: TLS_AES_256_GCM_SHA384
Server Temp Key: ECDH P-256 256 bits
Application protocol: h2
progress: 10% done
progress: 20% done
progress: 30% done
progress: 40% done
progress: 50% done
progress: 60% done
progress: 70% done
progress: 80% done
progress: 90% done
progress: 100% done

finished in 19.36s, 516.52 req/s, 94.12KB/s
requests: 10000 total, 10000 started, 10000 done, 0 succeeded, 10000 failed, 0 errored, 0 timeout
status codes: 0 2xx, 0 3xx, 10000 4xx, 0 5xx
traffic: 1.78MB (1866000) total, 148.44KB (152000) headers (space savings 88.12%), 1.44MB (1510000) data

min max mean sd +/- sd
time for request: 16.23ms 13.59s 5.54s 767.38ms 95.10%
time for connect: 2.23s 19.30s 4.85s 1.74s 65.20%
time to 1st byte: 6.57s 19.32s 10.39s 2.03s 61.90%
req/s : 0.52 1.52 1.00 0.19 61.40%
```

III - Client Hello Extensions

Several extensions are already present on the Hello client. For example, we have the cookie

```
struct{  
    opaque cookie <1..2 ^ 16-1>;  
} Cookie;
```

This allows you to prevent Dos type attacks by checking the reachability of the IP address that you claim to have.

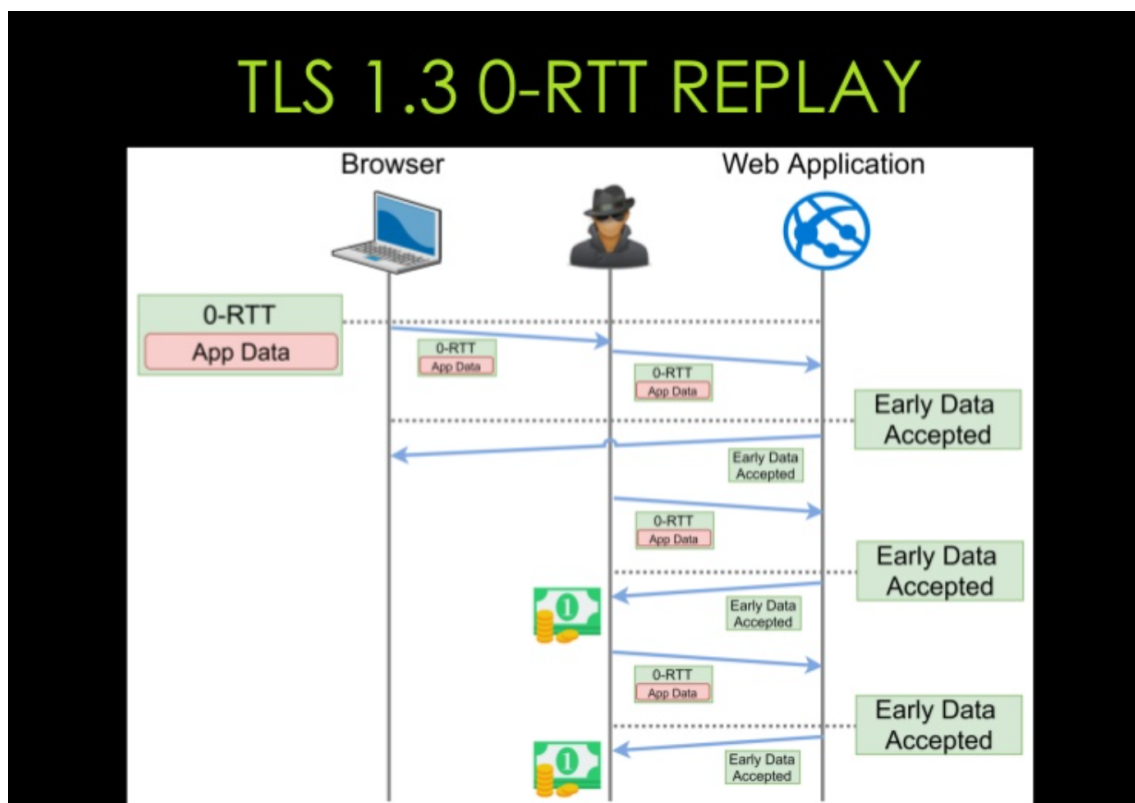
Note that this type of protection is also present in the QUIC protocol by the STK field: the goal is the same: to check that we have the IP address that we claim to have.

We also have extensions allowing the client to mount which versions of TLS are supported, which signature algorithms are supported, which CAs are accepted,

Thus, the extensions can allow either to add an optional service (help the server to select such certificate among all those it has), and others allow you to add a security service (the cookie to limit DoS attacks).

Now it's time to offer other extensions to the Hello client. For this, we looked at 0-RTT. This mode, very controversial, allows a client to send at the same time as the client hello, application data encrypted by the PSK of the last session. The time saving is very important: 1 RTT is the main selling point of this mode.

However, this increase in performance comes at the cost of weakened security. Indeed, these 0-RTTs are vulnerable to replay type attacks. Here is one:



An attacker goes into MITM, the client sends a 0-RTT that the attacker lets through. The server responds Early Data Accepted that the attacker forwards to the client to give him the illusion that there was no problem. Subsequently, the attacker sends the same 0-RTT several times to the server, which then retreats the same request. The attacker blocks the following Early data accepted.

These App Data requests can be \$ 10,000 POSTs, so the customer would start paying more than expected.

The problem is that the latest version of TLS still does not support an anti-replay mechanism for 0-RTT! Indeed, in draft 28 of the IETF, we can read:

“IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. This data is **not forward secret**, as it is encrypted solely under keys derived using the offered PSK.
2. There are **no guarantees of non-replay between connections**. Protection against replay for ordinary TLS 1.3 1-RTT data is provided via the server's Random value, but 0-RTT data does not depend on the ServerHello and therefore has weaker guarantees. This is especially relevant if the data is authenticated either with TLS client authentication or inside the application protocol. The same warnings apply to any use of the `early_exporter_master_secret`. ”

The 0-RTT is not forward secret by nature of the 0-RTT: we use a key that we already have. In addition, there are no guarantees of 0-RTT non-replay between connections.

We rely on this to offer extensions to the hello client:

1) Extension replay protection

We offer an extension to the Hello client **replay protection** which will serve as a field allowing the server to know if this 0-RTT is a replay or not.

For this, we already add another Client Hello extension which is: **anti-replay policy**. Allowing to indicate to the server the chosen anti-replay policy. This field can take several values:

- Single Use Ticket:

Authorize a PSK only once. The server remembers all valid PSKs and destroys them as soon as they are used once. The server must also implement an expiry date for these PSKs.

- Hello Recording Client

Derive from the Hello Client and the PSK a single value to reject duplicates. We can offer

$$CH_{\text{recording}} = \text{clientHello.random} \parallel H(\text{clientHello.random}, \text{PSK})$$

The server then keeps a strike register to keep in memory if a value has already been used. It is difficult for an attacker to produce a CHrecording because he is not supposed to know the PSK. However, be careful, the CHrecording space must be large enough to avoid anniversary type collisions!

In addition, it is complicated for the server to keep all these CHrecordings in memory, which could also be an attack vector: saturating the server's memory by sending several CHrecordings. Draft 28 proposes, without actually implementing it, that the server only keeps the CHrecordings in memory on a recording window. The server calculates the estimated reception time and if it is outside the window, it refuses it.

I do not completely agree with this proposition because even in a short window of time, an attacker can send multiple CHrecordings to saturate the server.

I prefer the proposal to implement Bloom filters on the server side: a so-called probabilistic and compact data structure ensuring no accepted replay but potentially refusing non-replay.

- Timestamp

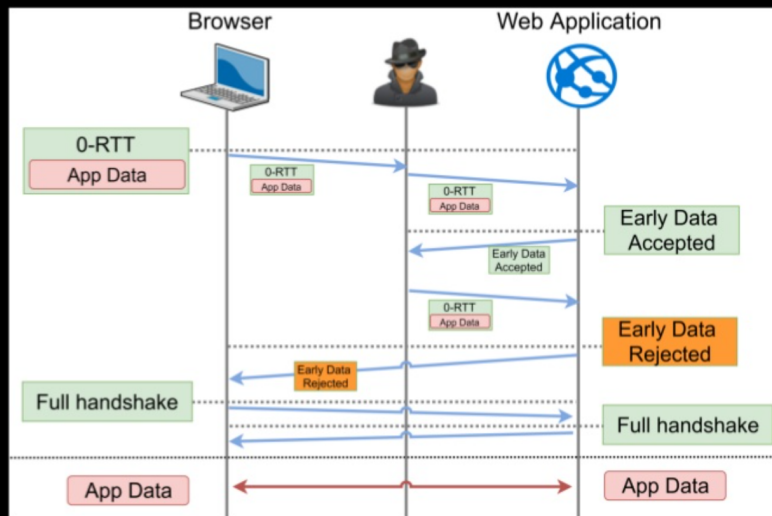
The client sends in the client Hello the time of sending. The server can refuse the 0-RTT if it is older than 200ms. However, attackers can be very quick. In addition, a server could refuse 0-RTTs when there was a simple congestion problem. The timestamp can be `send_time || H (send_time, PSK)`.

We have proposed 3 possible values for the anti-replay policy extension. For each of these policies, the anti-replay protection field is adequately filled (the ticket in the case of Single-use ticket, the CHrecording in the case of Client Recording, and a timestamp in the case of the timestamp).

2) Extension rejected_reason

replay_protection The extension is necessary, but not sufficient to stop the attacks based on the replay. Indeed, let's see the following attack:

UNIVERSAL REPLAY ATTACK



The client sends a 0-RTT to the server which accepts it. An MITM attacker intercepts the early data accepted and replay the 0-RTT.

The server refuses this one because it has already received it, so it sends the Early_data rejected to the client, which forces it to redo a whole new complete handshake.

Then, the client, believing that his 0-RTT sent has been refused, decides to resend the Application content of the 0-RTT.

The server accepts and has therefore processed the same application content twice!

First, we can say that this attack shows that the security of 0-RTT must be ensured at the application level. This does not respect the independence of the OSI layers, but is necessary.

However, I suggest the following: the server, in its early_data_rejected, could explain to the client why it refused 0-RTT. If the server refused it because it detected a replay in one way or another, the server must absolutely tell it so that the client does not return the same application content afterwards!

This could be defined as follows in the attack:

- The client sends a 0-RTT to the server with the extension to CHhello: require_rejected_reason: True and specifying the PSK_binder = PSK_binder_c
- The server responds with an Early_data_accepted which is intercepted by the attacker
- The attacker sends the 0-RTT back to the server.
- The server responds
 - Early_data_refused
 - > cause: already received
 - Already_received_psk_binder
 - > PSK_binder_c
 to the client
- Attacker blocks and sends a modified 0-RTT to the server posing as the client

- The server responds
 - Early_data_refused
 - > cause: failed to decrypt
 - Already_received_psk_binder
 - > PSK_binder_c
- A moment , the attacker lets pass towards the client.
- The client understands that their 0-RTT was accepted once, an attacker tried to play it back to the server, and then tried to send a spoofed 0-RTT pretending to be the client. The client therefore performs a full handshake and does not return the application content already sent in the 0-RTT. Or, it continues with the rest of the application content without doing a handshake. On receipt of the new application content, the server can then forget that there has been replay and not have to indicate it again if there is still a replay in a future 0-RTT.

The extension to the hello client would therefore be: **require_rejected_reason (true or false)**. If the value is true, the server must, in its response to 0-RTT, indicate the **reason for rejecting** 0-RTT if this happens as well as the PSK binder of the replayed message **if there has been replay since the reopening** of the session by 0-RTT.

The customer would thus have the choice, either to continue with the rest of the application content, or to perform a new full handshake, but not to return the application content already received.

This cause of rejection can be: already received, session ticket timeout, failed to decrypt, .. These causes of rejection could be coded like HTTP codes (200 OK, 404 page not found, 401 user not authenticated, ..).

Conclusion:

We were able to verify through an in-depth study of the handshake and a mid-scale benchmarking that the TLS1.3 protocol constitutes a major advance in terms of performance and services offered compared to TLS1.2. However, the increase in performance often comes at the cost of weakened security. We are offering extensions to the Hello client to address this.

Bibliography:

<https://tools.ietf.org/html/rfc8446>

<https://www.davidwong.fr/tls13/>

<https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/>

<https://nghttp2.org/>

<https://community.centminmod.com/threads/caddy-0-11-5-tls-1-3-http-2-https-benchmarks-part-1.16779/>

<https://fr.slideshare.net/cisoplatfrom7/playback-a-tls-13-story>