Real-world post-quantum TLS in Wireshark

Tuesday June 18th, 2024

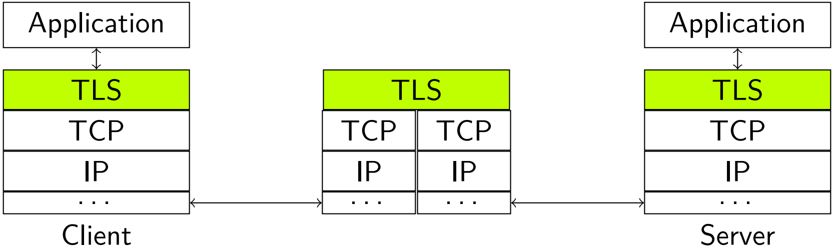
Peter Wu
Wireshark Core Developer
peter@lekensteyn.nl

About me

- Wireshark contributor since 2013, core developer since 2015.
- Areas of interest: TLS, QUIC, HTTP/3, Lua, security, . . .
- Cloudflare Research team. Recently worked on rolling out post-quantum TLS.

Transport Layer Security (TLS)

- Standard for securing network traffic. Web (HTTP), e-mail, databases, etc.
- Provides secure communication channel between two endpoints (client and server).
- Network protocol with two components:
  - Handshake Protocol: exchange capabilities, establish trust and establish keys.
  - Record Protocol: carries messages and protects application data fragments.



Post-quantum cryptography (PQC)

- Powerful quantum computers are expected in 15 to 40 years.[1]
- Essentially all Internet traffic today can be decrypted by these.
- Post-quantum (PQ) cryptography was designed to be secure against this threat.
- In active development: US National Institute of Standards and Technology (NIST) is almost done standardizing the initial post-quantum public-key algorithms.

[1] `https://blog.cloudflare.com/post-quantum-for-all/`

TLS decryption in Wireshark using the TLS key log file

- Text file with unique per-session secrets[2].
- TLS 1.2 format: `CLIENT_RANDOM` *`<Client Hello Random> <master secret>`*
- TLS 1.3 requires four different secrets (handshake and traffic secrets).

```
CLIENT_RANDOM 607AAA3D657D8A08F1073AE75B62CD284C87BB5504D275631CA86533707FB080 B27567070A3832CA2C072D1D0905647EF364C1E017A33001ED0BB2E
CLIENT_HANDSHAKE_TRAFFIC_SECRET e27a03ae85ae8035b331a1af6089dd1e2f300cce131b03fdb9f07a25f1a10876 8ac2e7e210e30e8f660048e20d45209935d6a
SERVER_HANDSHAKE_TRAFFIC_SECRET e27a03ae85ae8035b331a1af6089dd1e2f300cce131b03fdb9f07a25f1a10876 21c21f13865944c2c411ed1a7271809834dbe
CLIENT_TRAFFIC_SECRET_0 e27a03ae85ae8035b331a1af6089dd1e2f300cce131b03fdb9f07a25f1a10876 0de57183beff9a8c43994f517fba1d79ca374bff53b2a
SERVER_TRAFFIC_SECRET_0 e27a03ae85ae8035b331a1af6089dd1e2f300cce131b03fdb9f07a25f1a10876 f26e64d69b8095bbcdcbd04d48f2f9d96aedc1abc6463
```

- Import these secrets to Wireshark: *Edit* → *Preferences*, *Protocols* → *TLS*, *(Pre)-Master-Secret log filename*. Or right-click packet, *Protocol Preferences*.
- Ensure *Protocol Preferences* → TCP → *Reassemble out-of-order segments* is set!

-
  ```
  tshark -otls.keylog_file:keys.txt -r some.pcapng
  -otcp.reassemble_out_of_order:TRUE
  ```

[2] File format at `https://www.ietf.org/archive/id/draft-ietf-tls-keylogfile-02.html`

Generate key log file

- Set environment variable `SSLKEYLOGFILE` **before** starting Firefox or Chrome. Programs will append secrets to a file at this location.
- Start capture **before** running the application to capture the whole TLS handshake.
- Firefox on Windows, create `start-fx.cmd` file, without quotes in the `set` line:

  ```
  set SSLKEYLOGFILE=C:\Users\User\Desktop\keys.txt
  start firefox
  ```

- Chrome on Windows, create a shortcut with:

  ```
  chrome --ssl-key-log-file="C:\Users\User\Desktop\keys.txt"
  ```

- One-liner for Linux and macOS, start Firefox or Chromium with a new profile:

  ```
  SSLKEYLOGFILE="$PWD/keys.txt" firefox -no-remote -profile /tmp/ff
  SSLKEYLOGFILE="$PWD/keys.txt" chromium --user-data-dir=/tmp/cr
  ```

- For macOS: `export SSLKEYLOGFILE="$PWD/keys.txt";`

  ```
  open -na Google\ Chrome --args --user-data-dir=/tmp/cr
  ```

- curl 7.58.0 built with OpenSSL supports it too. (Not on macOS.)

Embed key log file in packet capture file

- TLS decryption requires pairing capture files with key log files. This makes switching between different files and file distribution more difficult.
- Solution: embed key log file in a **pcapng** file. Decryption Secrets Block (DSB).
- `editcap --inject-secrets tls,keys.txt in.pcap out-dsb.pcapng`
- Replace secrets: `editcap --discard-all-secrets --inject-secrets ...`
- inject-tls-secrets.py: script to embed a subset of TLS secrets in a pcapng file.[3]
  Example: given keys.txt and some.pcap, create some-dsb.pcapng:
  `./inject-tls-secrets.py keys.txt some.pcap`
- Since Wireshark 4.2: *Edit* → *Inject TLS Secrets*.

[3] https://gist.github.com/Lekensteyn/f64ba6d6d2c6229d6ec444647979ea24

Symmetric-key cryptography

- Symmetric encryption: sender and receiver have the same secret key.
- Authenticated Encryption with Additional Data (**AEAD**) added in TLS 1.2: AES-GCM, ChaCha20-Poly1305.
- Legacy (TLS $\leq$1.2): combine ciphers such as AES-CBC or RC4 with a Hashed Message Authentication Code (HMAC): HMAC-SHA256, HMAC-SHA1.
- Modern symmetric encryption is already post-quantum secure.

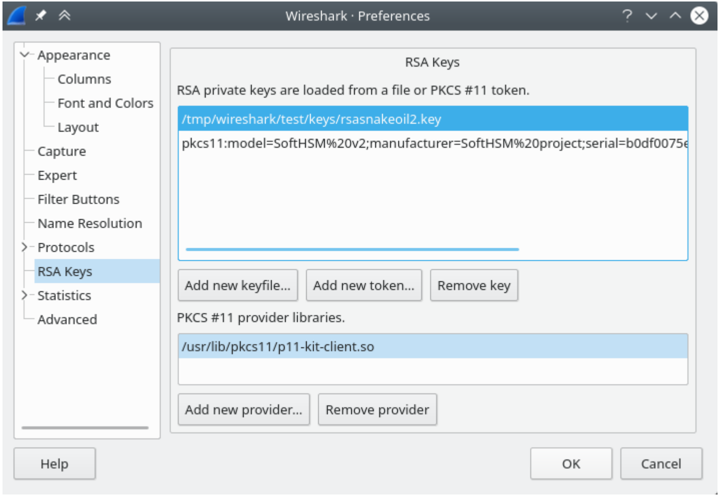
Public-key cryptography

- Public-key cryptography: different private and public key. Private encryption/signing key. Public decryption/verification key.
- Digital signature algorithms: RSA, **ECDSA**.
- Key agreement or key exchange (KEX): RSA (encrypt premaster secret against server key), **ECDHE** (Elliptic Curve Diffie-Hellman with ephemeral keys).
- Classical signature and key agreement algorithms are not PQ-secure.

RSA key exchange (TLS 1.2 and older)

- Client generates random premaster secret and encrypts it using server certificate.
- Server decrypts it using the RSA private key matching the certificate.
- Not forward secret. A single private RSA key file can decrypt all recorded traffic.
- Limitations:
  - Requires server admin to provide the key file.
  - Requires **TLS_RSA**_WITH_AES_128_CBC_SHA ciphers, not TLS_EC**DHE_**...
  - Does not work with session resumption.
  - Does not work with TLS 1.3.
- Example with SSL 3.0 (2006): rsasnakeoil2.pcap and rsasnakeoil2.key.

RSA Keys configuration



RSA Keys

RSA private keys are loaded from a file or PKCS #11 token.

/tmp/wireshark/test/keys/rsasnakeoil2.key

pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;serial=b0df0075e

PKCS #11 provider libraries.

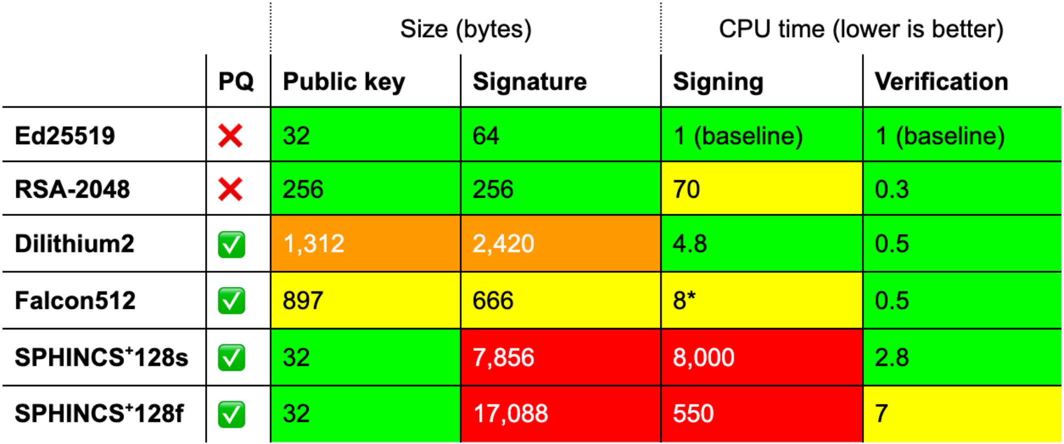
/usr/lib/pkcs11/p11-kit-client.so

- Not to be confused with *(Pre)-Master-Secret log filename*.
- Accepts PEM-encoded or PKCS#12 key file.
- PKCS#11 token and HSM support.
- `tshark -ouat:rsa_keys:'"rsa.key","password"'`

(EC)DHE key exchange

- Client generates new DH keypair, sends public DH key to server.
- Server generates new DH keypair, sends public DH key to client. Server **signs** it using private RSA/ECDSA key matching the certificate.
- Each side combines their own private key with the peer public key: shared secret.
- Each side throws away their ephemeral DH private key for perfect forward secrecy.
- Works with all TLS versions, including TLS 1.3. Example: tls12-dsb.pcapng
- Diffie-Hellman key exchange and RSA/ECDSA signatures are not PQ-secure.

Rough comparison of classical and post-quantum signature algorithms

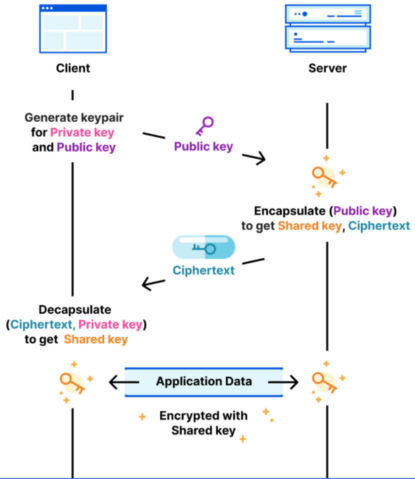


		Size (bytes)		CPU time (lower is better)	
	PQ	**Public key**	**Signature**	**Signing**	**Verification**
Ed25519	❌	32	64	1 (baseline)	1 (baseline)
RSA-2048	❌	256	256	70	0.3
Dilithium2	✅	1,312	2,420	4.8	0.5
Falcon512	✅	897	666	8*	0.5
SPHINCS$^+$128s	✅	32	7,856	8,000	2.8
SPHINCS$^+$128f	✅	32	17,088	550	7

Source: `https://blog.cloudflare.com/nist-post-quantum-surprise/` (2022)

Key agreement: post-quantum KEM vs classical DH

Key Encapsulation Mechanism (KEM)



Client

Server

Generate keypair for Private key and Public key

Public key

Encapsulate (Public key) to get Shared key, Ciphertext

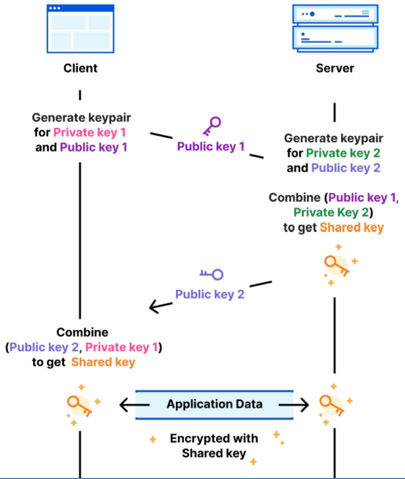
Ciphertext

Decapsulate (Ciphertext, Private key) to get Shared key

Application Data

Encrypted with Shared key

Diffie–Hellman (DH)



Client

Server

Generate keypair for Private key 1 and Public key 1

Public key 1

Generate keypair for Private key 2 and Public key 2

Combine (Public key 1, Private Key 2) to get Shared key

Public key 2

Combine (Public key 2, Private key 1) to get Shared key

Application Data

Encrypted with Shared key

Post-quantum key agreement

- Hybrid key agreement: Combine shared secrets from classic ECDHE (X25519) and post-quantum Kyber768 draft version.
- At least as secure as current X25519 deployments.
- Kyber is the basis for the future NIST FIPS 203 standard, *Module-Lattice-Based Key-Encapsulation Mechanism* (ML-KEM)[4].

[4] Initial Public Draft: `https://csrc.nist.gov/pubs/fips/203/ipd` (2023)

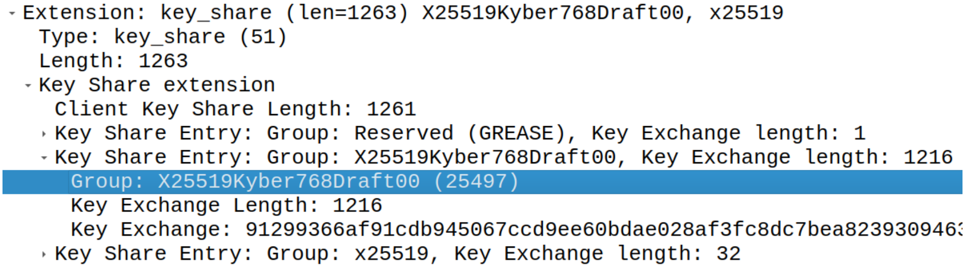
Rough comparison of classical ECDHE and post-quantum KEM

	Public key size		CPU time	
Group name	Client	Server	Client	Server
ECDHE: X25519	32		1 (baseline)	
ECDHE: NIST P-256	65		3.25	
ECDHE: NIST P-384	97		50.4	
ECDHE: NIST P-521	133		116.7	
PQ: Kyber768	1184	1088	5.53	3.53
Hybrid: X25519Kyber768Draft00	1216	1120	6.53	4.53

- Lower CPU time is better.
- Note: optimized Kyber768 versions are even faster than P-256.

TLS Group identifiers in Key Share and Supported Groups extensions

		Public key size	
Group name	Group ID	Client	Server
X25519	29, 0x001d	32	
NIST P-256	23, 0x0017	65	
NIST P-384	24, 0x0018	97	
NIST P-521	25, 0x0019	133	
X25519Kyber768Draft00	25497, 0x6399	1216	1120



```
▾ Extension: key_share (len=1263) X25519Kyber768Draft00, x25519
    Type: key_share (51)
    Length: 1263
  ▾ Key Share extension
      Client Key Share Length: 1261
    ▸ Key Share Entry: Group: Reserved (GREASE), Key Exchange length: 1
    ▾ Key Share Entry: Group: X25519Kyber768Draft00, Key Exchange length: 1216
        Group: X25519Kyber768Draft00 (25497)
        Key Exchange Length: 1216
        Key Exchange: 91299366af91cdb945067ccd9ee60bdae028af3fc8dc7bea823930946
    ▸ Key Share Entry: Group: x25519, Key Exchange length: 32
```

Real-world example: Google Chrome and Firefox versus Cloudflare

- Servers:
  - Cloudflare enabled PQ KEX in 2022 (about 20% Internet), see `https://pq.cloudflareresearch.com` or try `https://wireshark.org`
  - Google enabled support server-side in 2023.
- Clients:
  - Google Chrome 124 (April 2024): enabled by default. See *TLS 1.3 hybridized Kyber support* at `chrome://flags/#enable-tls13-kyber`.
  - Mozilla Firefox 124 (March 2024): set `security.tls.enable_kyber` to `true` via `about:config`. For QUIC, `network.http.http3.enable_kyber` (FF 128).
- `https://lekensteyn.nl/files/captures/chromium119-dsb.pcapng`
- `https://lekensteyn.nl/files/captures/firefox127-pq-dsb.pcapng`

What to look for

- Locate Client and Server Hello messages: `tls.handshake.type in {1, 2}`
- For PQ KEX, both client and server TLS extensions must have:
  - Supported Versions with TLS 1.3.
  - Supported Groups with X25519Kyber768Draft00 (25497).
  - Key Shares with X25519Kyber768Draft00.
- QUIC: runs over UDP instead of TCP. Uses TLS 1.3 for security.
- Match TLS Server Name with: `tls.handshake.extensions_server_name`
- Use stream index for linking packets via Custom column:
  - `tcp.stream or quic.connection.number or udp.stream`
- Use Ctrl + , and Ctrl + . to move to the previous/next packet in a conversation.

Common problems during PQ deployment

- Client or server were not properly configured with PQ support.
- TLS 1.3 is not enabled or TLS 1.2 or older is forced.
- The wrong server software was targeted by the client.
- An intercepting TLS middlebox was in use that did not support PQ.
- Bug in servers causing TCP resets for large Client Hello: `https://tldr.fail/`
- Bug in Rustls servers with Hello Retry Request.

Problem: Maximum Transmission Unit (MTU) issues

- Maximum Transmission Unit (MTU): typically 1500 for Ethernet. Can be lower due to tunneling/VPN overhead.
- Client connects, but during the TLS handshake times out waiting for the server.
- Client capture shows that the TCP handshake succeeds, but
  - Case 1: TLS Client Hello is sent, but never ACKed.
  - Case 2: TLS Server Hello is partially returned.[5] Check TCP sequence numbers.
- Solution: reduce MTU or apply TCP Maximum Segment Size (MSS) clamping.

[5] https://lekensteyn.nl/files/captures/tls-server-mtu-issue.pcap

Problem: non-compliant Hello Retry Request (HRR) handling

- If a TLS 1.3 server prefers a different key exchange group, it can send a Hello Retry Request (HRR).
- Client receives a TLS alert (Illegal Parameter) during the TLS handshake.
- Affects servers written in the Rust programming language using rustls.[6]
- Fixed in rustls 0.20.9 and 0.21.7 (August 2023).
- Servers must copy client Session ID into HRR to simulate TLS 1.2 session resumption for *middlebox compatibility mode*.
- `https://lekensteyn.nl/files/captures/time-hrr-rustls-bug.pcapng`

[6] `https://github.com/rustls/rustls/issues/1424`

Using Hello Retry Request to try another key exchange algorithm

- Cloudflare requests to origin servers supports PQ.[7]
- It can directly send the PQ key share (“preferred mode”).
- Or advertise PQ support, but initially send X25519 (“supported mode”).
- The latter can trigger a Hello Retry Request to ask the client to retry with the PQ key share. Adds one extra roundtrip.
- `https://lekensteyn.nl/files/captures/pq-origin-dsb.pcapng`

[7] https://blog.cloudflare.com/post-quantum-to-origins/

Active interception as alternative

- Previous methods were passive, they preserve the client-server behavior.
- Decryption without modifying workstations or smartphones requires active interception, an man-in-the-middle (MITM) attack.
- Caveat: active interception can affect the investigation. Different TLS parameters can be negotiated, TLS Client Authentication (mutual TLS) breaks, HTTP headers can change, certificate pinning result in new failures.
- Client talk to a proxy server which terminates TLS. The proxy starts a new TLS connection with the original server and forwards re-encrypted traffic.
- Typically a custom Root Certificate Authority (CA) certificate is installed on clients. Middlebox uses the corresponding CA private key to generate new certificates on-the-fly and serve these to clients.
- See `https://mitmproxy.org/`. Supports SSLKEYLOGFILE too!

Conclusion

- Post-quantum cryptography is here to protect data in the future.
- Use a key log file to enable TLS decryption in Wireshark.
- Embed these secrets in a pcapng file for easier distribution.
- Use the latest Wireshark version for the best results.
- See current PQ adoption on Cloudflare Radar and `https://pq.cloudflareresearch.com`
- For a more detailed background and key extraction from other applications, see `https://lekensteyn.nl/files/wireshark-ssl-tls-decryption-secrets-sharkfest18eu.pdf`

✉ peter@lekensteyn.nl
🌐 lekensteyn.nl
@Lekensteyn@infosec.exchange
🐦 @Lekensteyn