Quantum Attack on TLS 1.3: A Harvest-Now, Decrypt-Later Step-by-Step Guide

**Introduction**  
Harvest-Now, Decrypt-Later (HNDL) attacks exploit the time gap between today’s encryption and tomorrow’s cryptanalysis ([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ) ([PQC & Crypto-agility: Protect Against Steal & Decrypt Later](https://www.cryptomathic.com/blog/how-to-protect-yourself-against-steal-now-decrypt-later)). In essence, an adversary intercepts encrypted data now, stores it, and waits until they have a powerful quantum computer (or other advanced means) to decrypt it later. Transport Layer Security (TLS) 1.3 – the encryption protocol securing most HTTPS web traffic, emails, and messaging – is designed to be secure **today**. It uses *forward secrecy*: ephemeral (one-time) keys that even an attacker who steals a server’s long-term key cannot use to decrypt past sessions ([TLS 1.2 vs. 1.3—Handshake, Performance, and Other Improvements](https://www.catchpoint.com/http2-vs-http3/tls1-2-vs-1-3)). However, *quantum computing* threatens that forward secrecy. Algorithms like Shor’s can solve the mathematical puzzles (like factoring and discrete logarithms) underpinning RSA and elliptic-curve Diffie–Hellman (ECDH) encryption ([PQC & Crypto-agility: Protect Against Steal & Decrypt Later](https://www.cryptomathic.com/blog/how-to-protect-yourself-against-steal-now-decrypt-later)). This means a sufficiently advanced quantum computer could retrieve TLS 1.3’s ephemeral private keys and unravel the session secrets, decrypting everything that was recorded ([Harvest Now, Decrypt Later – Fact or Fiction? | Entrust](https://www.entrust.com/blog/2023/11/harvest-now-decrypt-later-fact-or-fiction)).

The stakes are high: **essentially all Internet traffic today could be decrypted by a future powerful quantum computer** (). Nation-state actors are **likely already stockpiling** intercepted ciphertext in bulk, anticipating the day (“Q-Day”) when quantum decryption becomes feasible ([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ) ([encryption - What is stopping someone from saving encrypted info, and decoding it later? - Cryptography Stack Exchange](https://crypto.stackexchange.com/questions/27728/what-is-stopping-someone-from-saving-encrypted-info-and-decoding-it-later)). In this article, we walk through a step-by-step quantum attack on TLS 1.3 using a HNDL approach – from massive data harvesting, through quantum key-cracking, to finally obtaining plaintext secrets. We focus on the **attacker’s process** (with minimal attention to defenses), illustrating how each phase of the attack works. Along the way, we’ll reference real-world surveillance programs and past cryptographic breaks as analogies, and use accessible language for readers with science or engineering backgrounds but less familiarity with TLS. By the end, you’ll understand how tomorrow’s quantum machines could retroactively expose today’s encrypted communications.

1. Capturing Encrypted TLS 1.3 Traffic (Data Harvesting)

The first step is **data harvesting**: the attacker quietly collects as much encrypted TLS 1.3 traffic as possible. This is *passive interception* – eavesdropping on communications without altering them. Attackers have several methods to achieve this:

**Packet Sniffing on Networks:** On local networks or tapped links, attackers use packet-capture tools like **Wireshark** or **tcpdump** to record raw traffic. These tools can log every byte transmitted, including the TLS handshake (e.g. Client Hello, Server Hello) and the encrypted Application Data packets. For instance, a hacker with access to a Wi-Fi router or an internet exchange point could run a sniffer and save all passing TLS packets to disk. Nothing in TLS 1.3 prevents an attacker from *recording* the ciphertext – it only prevents them from reading it without the keys.

**Network Taps at Infrastructure:** Nation-state actors and advanced threat groups may install fiber-optic taps or other interception appliances at **Internet backbone nodes** (e.g. ISP routers, submarine cable landing stations). A famous example is the NSA’s and GCHQ’s programs that tapped transatlantic fiber cables, ingesting **all** data flowing through. GCHQ’s “Tempora” program reportedly *“stores up to 100 petabytes of transatlantic data for at least three days”*, pulling in *“a secret feed of all internet and telephone traffic from more than 200 fibre optic cables”* ([GCHQ taps ALL transatlantic network traffic, says investigative journalist Duncan Campbell](https://www.computing.co.uk/news/2277868/gchq-taps-all-transatlantic-network-traffic-says-investigative-journalist-duncan-campbell)) ([GCHQ taps ALL transatlantic network traffic, says investigative journalist Duncan Campbell](https://www.computing.co.uk/news/2277868/gchq-taps-all-transatlantic-network-traffic-says-investigative-journalist-duncan-campbell)). Such systems can passively duplicate massive amounts of traffic in real time. The encrypted TLS 1.3 streams from hundreds of thousands of users can be siphoned wholesale and funneled into storage. Government agencies justify keeping encrypted data **indefinitely** if it’s for cryptanalysis purposes ([Leaked NSA Doc Says It Can Collect And Keep Your Encrypted ...](https://www.forbes.com/sites/andygreenberg/2013/06/20/leaked-nsa-doc-says-it-can-collect-and-keep-your-encrypted-data-as-long-as-it-takes-to-crack-it/)) – essentially hoarding today’s ciphertext on the chance they can crack it in the future.

**Compromised Routers and Switches:** Attackers might also compromise critical network hardware. By hacking an internet router (through malware or backdoors), they can turn it into a spy device that *sniffs* and forwards copies of traffic. For example, the recently discovered **Hiatus** malware on business-grade routers installed a modified tcpdump utility to capture passing packets ([New HiatusRAT router malware covertly spies on victims - Lumen Blog](https://blog.lumen.com/new-hiatusrat-router-malware-covertly-spies-on-victims/)) ([New HiatusRAT router malware covertly spies on victims - Lumen Blog](https://blog.lumen.com/new-hiatusrat-router-malware-covertly-spies-on-victims/)). Similarly, malware like **Cuttlefish** infects routers to *“monitor data that passes through them”* and even sets up a covert tunnel to exfiltrate that data ([New Cuttlefish malware infects routers to monitor traffic for credentials](https://www.bleepingcomputer.com/news/security/new-cuttlefish-malware-infects-routers-to-monitor-traffic-for-credentials/)) ([New Cuttlefish malware infects routers to monitor traffic for credentials](https://www.bleepingcomputer.com/news/security/new-cuttlefish-malware-infects-routers-to-monitor-traffic-for-credentials/)). These router compromises effectively create your own private “wiretap” on all traffic through that device – including TLS 1.3 connections, which the malware can quietly dump to a file. Attackers controlled by nation-states or cybercriminal cartels have been caught using such tactics to gather sensitive data in transit.

**Malware on Endpoints:** Yet another avenue is infecting one of the endpoints (client or server). If an attacker plants malware on a user’s computer or phone, that malware could be instructed to **silently capture all network traffic** before it leaves the device. For instance, a Trojan on a PC might hook into the operating system’s network stack to log every TLS connection’s bytes (handshake and encrypted content) and send it to the attacker’s server. Unlike active man-in-the-middle malware, this sniffer malware doesn’t need to break encryption; it just records the encrypted bytes as they are sent or received. This is effectively the same as using Wireshark on one’s own machine – except here the user is unwittingly doing it for the attacker. Similarly, on the server side, a compromised server could log all TLS traffic it handles. Many advanced persistent threats (APTs) have modules for network capture, as this can later yield treasure troves of intel once decryption is possible.

**Storing Massive Amounts of Encrypted Data:** Once captured, the encrypted TLS 1.3 traffic has to be stored until decryption is possible. Attackers assemble **automated, large-scale storage systems** for this purpose. Data can be aggregated into large PCAP (packet capture) files or fed into big-data pipelines. Given the volume (potentially petabytes from backbone taps), efficiency is key. Attackers might **filter and index** data as they store it – for example, storing only the TLS handshake messages (which are small) in one database, and the bulk encrypted payloads in another. They might compress or deduplicate data (many packets are repeats or uninteresting). NSA analysts have joked that *“certain three-letter institutions build large data centers... waiting for the first large quantum computer”* to decrypt it ([encryption - What is stopping someone from saving encrypted info, and decoding it later? - Cryptography Stack Exchange](https://crypto.stackexchange.com/questions/27728/what-is-stopping-someone-from-saving-encrypted-info-and-decoding-it-later)). Indeed, leaked documents suggest agencies have been empowered to *“collect and indefinitely keep any encrypted communication”* for future exploitation ([Leaked NSA Doc Says It Can Collect And Keep Your Encrypted ...](https://www.forbes.com/sites/andygreenberg/2013/06/20/leaked-nsa-doc-says-it-can-collect-and-keep-your-encrypted-data-as-long-as-it-takes-to-crack-it/)). In practice, an attacker might store each intercepted TLS session with an ID and timestamp, archive the ciphered data to cloud storage or tape, and maintain an index so it can be retrieved when needed. The goal is to build a **time capsule of ciphertext** – a database of secrets locked by today’s cryptography, but ready to be unlocked by tomorrow’s cryptanalysis.

At this stage, everything the attacker has is **still encrypted**. They cannot read the content yet. But they have achieved a critical prerequisite for HNDL: they possess a copy of the TLS 1.3 sessions (handshakes and encrypted data) that they want to eventually decrypt. Now the attacker moves on to organizing this trove for future key cracking.

2. Identifying & Organizing TLS 1.3 Sessions

After hoarding raw packet captures, the attacker needs to make sense of them – separating and identifying individual TLS **sessions** (i.e. individual connections between a client and server). TLS 1.3 traffic, as seen on the wire, is a stream of TCP packets that include handshake records and application data records. Simply storing millions of packets isn’t useful unless the attacker can organize them by session. This is akin to sorting a huge pile of puzzle pieces into individual puzzles. Here’s how attackers structure and index captured TLS 1.3 data:

* **Session Separation via Handshakes:** The **Client Hello** and **Server Hello** messages mark the start of a TLS session. An attacker’s processing software will scan the captured data for these handshake messages. For each Client Hello, it creates a new session entry. The Client Hello is easy to spot: it’s a TLS record (content type 22) with the byte pattern that indicates a handshake and a “client\_random” value. Likewise, the Server Hello (the server’s reply) confirms that session. By parsing these, the attacker can delineate one session from the next. Each session can be labeled with a unique identifier. Often, the client’s 32-byte random value (in the Client Hello) can serve as a unique ID for the session – it’s essentially a nonce that should be unique per handshake ([A Walkthrough of a TLS 1.3 Handshake](https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art080)). In older TLS versions, there was also a “Session ID” field that could be used to identify resumed sessions; in TLS 1.3 this field is present for compatibility but is not used in new session negotiations ([A Walkthrough of a TLS 1.3 Handshake - Command Line Fanatic](https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art080)) ([What is the equivalent of SSLSessionID for TLS1.3 sessions](https://stackoverflow.com/questions/60115179/what-is-the-equivalent-of-sslsessionid-for-tls1-3-sessions)). So the attacker focuses on the random values and the 5-tuple (client IP, client port, server IP, server port) to identify flows.
* **Indexing Key Metadata:** For each session, the attacker will store key metadata from the handshake:
  + **Timestamp**: when the handshake occurred (from packet timestamps).
  + **Client and Server IP addresses** (and ports) for context.
  + **Protocol details**: TLS version (likely 1.3 for our case), chosen cipher suite, and key exchange method. For example, the handshake might negotiate TLS 1.3 with cipher suite TLS\_AES\_128\_GCM\_SHA256 and ECDH X25519 key exchange. This info is in the Server Hello (cipher suite) and the Key Exchange extension.
  + **Ephemeral Public Keys**: crucially, the attacker logs the actual public key values from the handshake’s key exchange (we’ll expand on this in the next section).
  + **Session Tickets or Identifiers**: If the server issued a session ticket (for resumption) or if the client offered a pre-shared key identity, those are noted too, to understand if a session was resumed. Resumed sessions might not have a full key exchange, so linking them to the original session is important.

All this data can be stored in a structured way. Think of a table where each row is one TLS session and columns include time, client IP, server IP (or server name), cipher suite, ephemeral key, etc. This makes it easier later to feed the relevant data (like the public keys) into quantum decryption algorithms.

* **Mapping Handshake to Application Data:** After the handshake, all TLS 1.3 **Application Data** records flow. The attacker must map those encrypted data packets to the correct session. Typically, this is done by the 5-tuple and sequence: all packets belonging to the same TCP connection (same source/dest IP and port, between the ClientHello and that connection’s FIN packet) are part of one TLS session. Attackers can use automated traffic analysis tools – similar to how network monitors or IDS systems reassemble flows. For instance, the tool Zeek (formerly Bro) can parse TLS handshakes and log each connection’s details, including server name (from SNI), ciphers, and even the elliptic curve used ([ssl.log — Book of Zeek (git/master)](https://docs.zeek.org/en/master/logs/ssl.html)). Passive observation systems can reliably identify that *“the server name is X, cipher is Y, elliptic curve is X25519”* for a given session ([ssl.log — Book of Zeek (git/master)](https://docs.zeek.org/en/master/logs/ssl.html)), even though they cannot see the content. Our attacker’s system would do the same, then tag all subsequent packets with the corresponding session ID.
* **Dealing with Encrypted Handshake Components:** TLS 1.3 encrypts much of the handshake after the initial key exchange. The server’s Certificate and CertificateVerify messages, for example, are encrypted with keys derived from the handshake secrets. At capture time, the attacker can’t read these (they appear as opaque blobs). However, this doesn’t stop session identification – the attacker doesn’t necessarily need to know the server’s identity or certificate at this point. They just need to group packets correctly. (Later, when they decrypt, those handshake details will become readable too.) One implication is that the attacker might not initially know *which website or service* a session was for, unless the Client Hello’s Server Name Indication (SNI) was present. SNI is sent in plaintext in TLS 1.3 (unless Encrypted Client Hello is used), and it reveals the target hostname (e.g., “[www.example.com”](http://www.example.xn--com-9o0a/)) ([TLS interception - ORG Wiki](https://wiki.openrightsgroup.org/wiki/TLS_interception)). Attackers will record SNI when available to help prioritize and classify sessions (for example, a session with SNI bankofamerica.com is clearly high-value). If no SNI (like some APIs), they might only have an IP address to go on initially.
* **Session Resumption and 0-RTT Tracking:** TLS 1.3 allows sessions to be resumed using a pre-shared key (PSK) from a previous session, and optionally clients can send 0-RTT data (data on the very first flight without waiting for a full handshake). These cases complicate session organization:
  + A **resumed session** (1-RTT) will have a ClientHello that includes a “pre\_shared\_key” extension referencing a ticket, and a abbreviated handshake (no certificate exchange, possibly even no new Diffie-Hellman if PSK-only mode). The attacker’s system must detect this and link the resumed session’s ticket identifier back to the original session where that ticket was issued. That way, when the original session is eventually decrypted, the attacker gains the PSK and can use it to derive keys for the resumed session as well. Automated correlation is needed: the session ticket is essentially an opaque blob to everyone except the server, but the attacker can use it as a unique label. They store “Session X resumed via Ticket Y” and know to process Session X and Y together later.
  + **0-RTT data** (early data) is sent by the client before the handshake completes, encrypted with keys derived from the PSK of a previous session. Attackers capturing 0-RTT will see application data records immediately after the ClientHello, even before ServerHello. These are encrypted with a key the client derived from the PSK. The server’s Finished message will indicate whether it accepted the 0-RTT. The attacker tags these early data packets and knows they can’t decrypt them until they have the PSK from the original session. Also, 0-RTT has no forward secrecy (it’s a replayable blob of data encrypted with a static key shared from before) – so if the original session’s key is obtained, the 0-RTT data can be decrypted as well. (It’s worth noting that 0-RTT comes with known security risks like replay attacks ([TLS 1.2 vs. 1.3—Handshake, Performance, and Other Improvements](https://www.catchpoint.com/http2-vs-http3/tls1-2-vs-1-3)), but here we care that it’s extra encrypted data to manage.)

In summary, during this phase the attacker turns raw captures into an **organized archive of TLS sessions**. Each session entry includes the handshake metadata (especially the public key exchange values and any identifiers) and pointers to the encrypted data segments. This organization is critical; without it, later steps (like using quantum computing on a specific session’s key exchange) would be like finding a needle in a haystack. By structuring the data now, the attacker sets the stage to systematically attack each session’s encryption when the capability becomes available.

3. Extracting & Matching Public Keys to TLS Session Data

The heart of breaking TLS 1.3’s encryption lies in the **ephemeral keys** used during the handshake. In this step, the attacker plucks out those keys – specifically, the *public* components of the Diffie–Hellman key exchange – from each stored session, and ensures they’re linked to the correct session records. Let’s break down how this works:

**Ephemeral Diffie–Hellman in TLS 1.3:** TLS 1.3 uses an ephemeral Diffie–Hellman (DH) key exchange (often over elliptic curves) to establish a shared secret between client and server. “Ephemeral” means new keys are generated for every session, providing forward secrecy. In a typical ECDH handshake: the client generates a random private key a and computes a public key A = g^a (on an elliptic curve or DH group), and the server does similarly with private b and public B = g^b. They exchange A and B, and each side can compute the shared secret g^{ab}. In TLS 1.3, this exchange is embedded in the handshake messages:

The **ClientHello** carries one or more *Key Share* entries (in the “key\_share” extension). Each entry includes the identifier of a DH group (for example, X25519, or a post-quantum scheme like Kyber) and the client’s ephemeral public key for that group. If the client offers multiple key shares, the server will choose one (commonly clients just send one to avoid bloat).

The **ServerHello** includes *its* chosen Key Share extension, with the server’s ephemeral public key in the selected group.

For example, if using X25519 (an elliptic curve Diffie–Hellman), the ClientHello’s key\_share might contain a 32-byte public value for X25519. The ServerHello will reply with its own 32-byte X25519 public value. All this is unencrypted and visible to the passive attacker. In a real TLS 1.3 handshake captured in Wireshark, you’d see something like “Key Share: Group X25519, Key Exchange: ”. The *Illustrated TLS 1.3* project shows that in the ClientHello, *“the client sends one or more ephemeral public keys... This allows the rest of the handshake after the ClientHello and ServerHello to be encrypted.”* ([The Illustrated TLS 1.3 Connection: Every Byte Explained](https://tls13.xargs.org/)). In the raw bytes, you can spot the extension 0x0033 (key\_share) and the length, group id, etc., followed by the client’s public key bytes ([The Illustrated TLS 1.3 Connection: Every Byte Explained](https://tls13.xargs.org/)). Likewise, the server’s Key Share extension is in its Hello; for X25519 it will similarly have the 32-byte pubkey. We can cite the example: *“The server sends a public key using the algorithm of the public key sent by the client...”* ([The Illustrated TLS 1.3 Connection: Every Byte Explained](https://tls13.xargs.org/)) – in that example, both sides used X25519 and we see the server’s public key bytes in the ServerHello.

The attacker’s automated parsers will extract these ephemeral **public keys** (A and B in our notation) from each session’s handshake messages. These values are absolutely key to the later decryption effort – they are what the attacker will feed into a quantum algorithm to derive the private keys. So the system might create a list or database of all captured key shares: e.g., Session 12345: Group=X25519, ClientPubKey = ..., ServerPubKey = ... (in hex). If a post-quantum algorithm like **Kyber** (a lattice-based key encapsulation mechanism) is used in a handshake, the concept is similar: Kyber involves the client sending a public key and the server responding with a ciphertext that encapsulates the shared secret. For simplicity, many experimental post-quantum TLS 1.3 variants still have public components from each side. The attacker would extract those as well (client’s PQ public key, server’s response).

**Matching Keys to Sessions:** It’s crucial that each public key gets correctly matched to its session and counterpart key. In a straightforward full 1.3 handshake, the client and server each provide one ephemeral public key. The attacker can pair them: in Session X, note *PublicKey\_client = A*, *PublicKey\_server = B*. These two together define the Diffie–Hellman exchange for that session. The attacker stores them together with the session’s identifier. If multiple key shares were offered (say, client offered X25519 and Kyber hybrid), the server picks one – the attacker sees which one is in ServerHello and focuses on that. Any unused client key shares can be ignored (they didn’t contribute to the secrets).

For **session resumption cases**:

If the session is purely PSK resumption **without** DH (PSK-only mode), the handshake will not have new DH key shares. That means there’s no ephemeral public key to extract – the session’s secrets come from the PSK (which was from an earlier session). In this case, the attacker must realize this and mark the session as requiring the original session’s key. The ClientHello’s “pre\_shared\_key” extension will indicate which ticket/PSK is being used. The attacker correlates that with the original session where that ticket was issued. Essentially, they note: *Session Y has no new DH keys; uses PSK from Session X.* So later, to decrypt Session Y, they will need the master secret from Session X. This tracking is part of the indexing from section 2.

If the session uses PSK **with DHE** (which TLS 1.3 allows for added security – a fresh DH on top of the PSK), then there *will* be new ephemeral keys in the handshake. The attacker will extract those same as any other full handshake. The presence of “psk\_dhe\_ke” mode is indicated in the ClientHello’s “PSK key exchange modes” extension ([The Illustrated TLS 1.3 Connection: Every Byte Explained](https://tls13.xargs.org/)). The attacker’s logic can handle this: even though it’s resuming, it’s effectively a new DH exchange with its own A and B, so treat it like a fresh key exchange to break (with the benefit that even if quantum fails to break the lattice or something, the attacker could still fall back to breaking the classical part).

**0-RTT complication:** In a 0-RTT scenario, the client sends data encrypted with a PSK key before getting the server’s response. The key for 0-RTT is derived from the PSK (from the old session). The attacker capturing it will log those early data packets but cannot decrypt them until they recover that PSK (by breaking the original handshake). The key exchange for the main handshake might still happen (if 0-RTT + DHE is used, which results in forward-secret 1-RTT keys for later data). So the attacker still extracts the DH keys from the ServerHello if present. The 0-RTT data is just tagged in storage. Once the original session is solved and PSK known, that key can be used to decrypt the 0-RTT messages (which are typically encrypted with a separate “early traffic secret”).

In short, the attacker now has, for each TLS 1.3 session of interest:

The **ephemeral public key(s)** exchanged (and knows which algorithm/group was used, e.g., secp256r1, X25519, or a PQ scheme like Kyber).

Knowledge of any sessions that didn’t have an exchange (PSK-only) and how to link them to an earlier exchange.

A complete map linking these keys to the stored ciphertext of the same session.

This collection of public keys is the “encrypted data’s Achilles’ heel” – but only if one can compute the corresponding **private keys** or shared secrets. With classical computers, ephemeral Diffie–Hellman is designed to be intractable to reverse (that’s the Diffie–Hellman problem / discrete log problem). *No amount of classical computing power today can feasibly derive the private key from the public key for well-chosen parameters.* For example, X25519 (ECDH on Curve25519) has a 256-bit exponent – brute-forcing that is astronomically hard. That’s why forward secrecy holds against eavesdroppers *today*. However, this is exactly where the advent of **quantum computing** changes the game. The attacker, having patiently harvested and indexed all these public keys, now moves to use quantum algorithms to retrieve the secrets behind them.

4. Decrypting TLS 1.3 Key Exchange with Quantum Computing

At some point – perhaps years or even decades after the data was captured – the attacker finally has access to a quantum computer powerful enough to break current cryptographic algorithms. This is the moment HNDL attackers have been waiting for. They will use quantum computing, specifically **Shor’s Algorithm**, to crack the TLS 1.3 key exchanges and obtain the ephemeral *private* keys (or shared secret) for each session ([Harvest Now, Decrypt Later – Fact or Fiction? | Entrust](https://www.entrust.com/blog/2023/11/harvest-now-decrypt-later-fact-or-fiction)) ([PQC & Crypto-agility: Protect Against Steal & Decrypt Later](https://www.cryptomathic.com/blog/how-to-protect-yourself-against-steal-now-decrypt-later)).

**Shor’s Algorithm in a Nutshell:** Peter Shor discovered in 1994 that a sufficiently large quantum computer can solve two mathematical problems exponentially faster than classical computers: integer factorization (breaking RSA) and discrete logarithms (breaking Diffie–Hellman, including elliptic-curve DH) ([PQC & Crypto-agility: Protect Against Steal & Decrypt Later](https://www.cryptomathic.com/blog/how-to-protect-yourself-against-steal-now-decrypt-later)). TLS 1.3’s security relies on the hardness of the elliptic-curve discrete log problem (for ECDH) or similar problems for other groups. For example, given an elliptic curve public key A = g^a (with g a base point and a the secret exponent), a classical attacker can’t feasibly find a. But a quantum attacker running Shor’s algorithm *can find a in polynomial time*. In practical terms, if the TLS handshake used **ECDH (secp256r1 or X25519)**, Shor’s algorithm can compute the server’s or client’s 256-bit private exponent from the 256-bit public key ([The elliptic curve discrete log problem is hard because there is no ...](https://www.reddit.com/r/badmathematics/comments/uaoi7g/the_elliptic_curve_discrete_log_problem_is_hard/)). If the handshake used finite-field DH (less common in TLS 1.3, but similar principle), Shor’s would solve that too. For RSA (which isn’t used in TLS 1.3 handshakes, but was in older TLS versions), Shor’s factors the RSA modulus to recover the private key ([Harvest Now, Decrypt Later – Fact or Fiction? | Entrust](https://www.entrust.com/blog/2023/11/harvest-now-decrypt-later-fact-or-fiction)). The consensus in the crypto community is that *“large-scale quantum computers will be able to break the majority of current asymmetric cryptographic standards”* ([PQC & Crypto-agility: Protect Against Steal & Decrypt Later](https://www.cryptomathic.com/blog/how-to-protect-yourself-against-steal-now-decrypt-later)). The timeline is debated, but estimates say within 15-20 years there’s a significant chance a quantum computer could break RSA-2048 in hours ([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ). Our attacker has been forward-thinking (or is a nation-state with early quantum capabilities), so now they unleash this power on the stored data.

**Quantum Cracking of ECDH Example:** Take one stored session using X25519 (a common curve in TLS 1.3). The attacker possesses the public key A = g^a mod p from the client (and B = g^b mod p from the server). The attacker can choose to target either one – say the server’s. They feed B (and the curve parameters) into their quantum computer running Shor’s algorithm. In relatively short order, the quantum computer calculates the server’s secret exponent b. Now the attacker has the server’s ephemeral **private key** for that session. With b in hand, they can compute the Diffie–Hellman shared secret: since they also have the client’s public key A, they compute A^b = (g^a)^b = g^{ab}. This g^{ab} is the **pre-master secret** for the TLS 1.3 session – essentially the raw shared secret that the client and server also computed during the handshake. (Alternatively, the attacker could have found a from A and then computed B^a, same result.) The key point is, the once-ephemeral secret ab is no longer secret.

If the TLS 1.3 handshake was a **hybrid post-quantum exchange** (for instance, some sessions might have used an experimental X25519+Kyber combination ()), the attacker would need to break *both* parts to get the full shared secret (assuming the scheme combines them securely). Shor’s algorithm can tackle the classical part (X25519) but **Kyber (a lattice-based KEM)** is designed to resist quantum attacks. There is *no known efficient quantum algorithm to break Kyber’s underlying problem*. So if our attacker encounters a session that used a quantum-resistant key exchange, they might be stuck – at least until/unless new quantum algorithms or mathematical breakthroughs emerge. For the purposes of our scenario, we assume most sessions used standard ECDH (since TLS 1.3 as of now doesn’t natively include PQC). The attacker could simply skip any PQC-protected sessions (they remain confidential) and focus resources on the vulnerable ones. It’s worth noting that early adoption of post-quantum crypto by some organizations is happening specifically to foil this kind of HNDL threat ([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ). But those are in the minority, meaning a treasure trove of “classical” ECDH handshakes remain for the taking.

Once the attacker has computed the shared Diffie–Hellman secret for a session, the next step is to derive the actual symmetric encryption keys from that. TLS 1.3 defines a **key schedule** that uses the HKDF (HMAC-based Key Derivation Function) to mix the shared secret with some salt and context (like handshake transcripts) to produce various keys (handshake keys, application traffic keys, etc.) ([RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3](https://datatracker.ietf.org/doc/html/rfc8446)) ([The TLS 1.3 key schedule. The values of context and label inputs (H](https://www.researchgate.net/figure/The-TLS-13-key-schedule-The-values-of-context-and-label-inputs-H_fig2_353594502)). But this is a purely deterministic, known algorithm. The attacker can perform the same steps that the client and server did:

They combine the DH shared secret g^{ab} with the indicated “DH extractor” function (HKDF-Extract) using a salt (which for the first stage might be the “zero” salt in TLS 1.3, since handshake secret = HKDF-Extract(salt=0, DHsecret)).

They get the **Handshake Secret**, then derive handshake traffic keys. With the handshake secret and knowledge of the handshake transcript, they derive the **Master Secret**, and from that the **Application Traffic Secrets** (separate for client and server) ([TLS 1.2 vs. 1.3—Handshake, Performance, and Other Improvements](https://www.catchpoint.com/http2-vs-http3/tls1-2-vs-1-3)) ([RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3](https://datatracker.ietf.org/doc/html/rfc8446)).

Finally, they derive the actual **encryption keys and IVs** for the application data.

All the needed ingredients for key derivation are known to the attacker at this point:

The shared secret (now obtained via quantum attack).

The full handshake transcript (they captured all handshake messages, and can now even decrypt the encrypted portions of the handshake since they have handshake keys).

The exact KDF and cipher suite information (from the metadata they stored).

So, for session X, the attacker runs a small program (not even on a quantum computer – a normal computer is enough for key derivation) to calculate the TLS 1.3 keys. If the cipher suite was, say, TLS\_AES\_128\_GCM\_SHA256, the output will be a 128-bit client write key, a 128-bit server write key, plus their IVs and handshake keys. They have effectively re-computed the **“session keys”** that were originally established in that TLS session.

It’s important to note: **forward secrecy has now been nullified for that session**. The TLS 1.3 design meant that even if someone got the server’s long-term private key, they couldn’t derive this session key (because of DH). But by breaking the DH itself, the attacker has achieved what was once thought computationally unfeasible. As the Open Rights Group wiki on TLS interception notes, without forward secrecy an eavesdropper who obtains the server’s private key (e.g., via a court order or theft) can decrypt past sessions ([TLS interception - ORG Wiki](https://wiki.openrightsgroup.org/wiki/TLS_interception)). With forward secrecy (DH), that was impossible – *until* a quantum computer enters the picture. Now, with quantum-derived DH secrets, retrospective decryption is possible even for sessions that used PFS. In essence, we’ve gone back in time and stolen the one-time padlock keys that secured those conversations.

At this stage, the attacker possesses the symmetric encryption keys for each targeted TLS session. The next step is to actually apply those keys to the stored ciphertext and recover the plaintext data – i.e., perform the decryption of the Application Data records.

5. Matching Recovered Session Keys to Encrypted Application Data

Having obtained the secret session keys for a given TLS 1.3 session, the attacker now faces a more straightforward but still non-trivial task: using those keys to decrypt the stored encrypted traffic for that session. This step is about **joining the keys with the data**.

Recall that in section 2, the attacker meticulously mapped each encrypted packet to its session. So for *Session ABC123* (for example, identified by a client random or some ID), they have:

A list of encrypted TLS records (Application Data records) captured.

The session’s client->server and server->client traffic keys and IVs (just derived via the quantum-broken handshake).

The details of the cipher (AES-GCM, ChaCha20-Poly1305, etc.) and the record protocol specifics (TLS 1.3 uses a AEAD scheme where each record has a sequential number used in nonce construction, etc.).

**Determining which key for which direction:** TLS uses separate keys for encryption in each direction. Typically, one key encrypts data sent by the client, and another key encrypts data sent by the server. The attacker will know which is which because the TLS key schedule outputs a “client application traffic secret” and a “server application traffic secret.” They can derive both. Now, how to tell which captured packet was client->server vs server->client? Usually, by network addresses: e.g., if the client was the one with IP 1.2.3.4 and the server 5.6.7.8, then packets from 1.2.3.4:port\_x to 5.6.7.8:443 are client-to-server, and those from 5.6.7.8:443 back to 1.2.3.4:port\_x are server-to-client. The stored metadata includes which IP was client and which was server. So the attacker can label the encrypted records as “use client key” or “use server key” accordingly.

**Using sequence numbers and nonces:** TLS 1.3’s record encryption is also dependent on a per-record nonce (initialization vector) typically derived from a combination of a fixed IV (derived from the traffic secret) and a per-record sequence number (incrementing). Because the attacker captured the records in order, they can replay the sequence. If any packets were lost or out of order, the record protocol’s sequence numbers might have gaps, but typically they can infer the order. Many implementations simply use an implicit sequence number starting at 0 for the first record. The first record encrypted with the client’s key would use nonce = IV ⊕ 0, the second uses IV ⊕ 1, and so forth (with some encoding). The attacker’s decryption tool will replicate this process: iterate through each record in the order it was sent, keep a counter, derive the correct nonce, then decrypt using the symmetric cipher and key.

**Automated decryption process:** With all pieces ready, the attacker can run an automated decryption for each session:

Fetch the list of encrypted records for session X.

For each record, determine direction (client or server).

Use the corresponding AES or ChaCha key and compute the nonce (which often involves the record sequence number XORed with the IV ([The TLS 1.3 key schedule. The values of context and label inputs (H](https://www.researchgate.net/figure/The-TLS-13-key-schedule-The-values-of-context-and-label-inputs-H_fig2_353594502))).

Decrypt the ciphertext and verify the authentication tag (AES-GCM includes an auth tag per record).

If the tag verifies, output the plaintext of that record.

This is essentially what Wireshark does if you feed it TLS session keys via a key log file. In fact, the attacker’s workflow at this point is similar to a debugging scenario: when developers debug TLS, they sometimes use the TLS key logging feature in browsers to get the symmetric keys and load them into Wireshark to decrypt traffic. Wireshark matches the keys to sessions using the ClientHello’s random value (CLIENT\_RANDOM label) and decrypts the flows (). The attacker is doing the same thing, but their keys came from quantum codebreaking rather than a browser’s debug output.

The attacker must ensure that the keys are matched to the correct data streams. If there was any mix-up (e.g., using the wrong session’s key on a ciphertext), decryption will fail (auth tags won’t match). Therefore, maintaining the integrity of the session mapping is critical. Fortunately, TLS 1.3’s design (with unique randoms and nonces per session) makes accidental cross-use of keys unlikely if the indexing was done properly.

One useful piece of information attackers leverage are **TLS record headers**. Each TLS record (even encrypted) has a header that includes:

Content type (in TLS 1.3, this is always “application\_data” (23) for encrypted payloads after handshake, and handshake (22) for initial messages, etc., but after handshake they often all appear as type 23 because even handshake messages get encrypted and encoded as application\_data).

Length of the record.

In TLS 1.3, the actual content type (like whether the plaintext was handshake or application) is encrypted and hidden at the end of the record. But the mere existence of records and their sizes can help confirm ordering.

The attacker might note that after the handshake, all records in that connection are application data records until possibly a “CloseNotify” alert at the end. So they know to use application traffic keys for them. (In TLS 1.2, different content types could appear in the clear header; TLS 1.3 simplified that, always showing 23). In any case, the mapping of keys to packets is straightforward with the connection context.

**Tagging for correct decryption:** The attacker’s decryption tools likely automatically tag or group packets by session ID. When the quantum step yields a new session key, the system can look up all packets tagged with that session ID and then attempt decryption. It might mark each packet as “decrypted” and store the plaintext or at least verify integrity. A successful decryption of the first few records (like the HTTP request’s beginning) will confirm that the correct key has been applied to the correct data.

Remember, the attacker also can now decrypt the *handshake* messages that were encrypted (like the Certificate, CertificateVerify, Finished). They might do that as well, to, for example, retrieve the server’s certificate. Why would they care? Knowing the server certificate tells them what site or service the data was from if SNI was missing. It could also give them long-term identity of the server. However, since our focus is on application data, this is a side benefit. Still, an intelligence agency might log, “Session X: now decrypted, it was a connection to gmail.com (got the certificate and see plaintext).”

By the end of this step, the attacker has transformed what was a pile of indecipherable TLS ciphertext into **reassembled plaintext streams**. Each TLS session’s encrypted application data is now a sequence of decrypted bytes, typically representing higher-level protocols like HTTP, SMTP, or API calls. Now the real value can be extracted from that plaintext content.

6. Decrypting the TLS Application Data & Extracting Sensitive Information

With the TLS application data decrypted, the attacker can **read the communications in plain form**, just as the legitimate participants did originally. This includes all the potentially sensitive information that was protected by TLS encryption – which is the ultimate prize of the HNDL attack. The process now becomes one of data analysis and extraction:

**Reassembling the Plaintext Streams:** TLS record boundaries don’t necessarily align with message boundaries in the application protocol. So the attacker likely needs to reassemble the continuous stream. For example, if it’s HTTPS, they will reconstruct the full HTTP request and response from the sequence of decrypted TLS records. Tools can combine the decrypted segments based on sequence numbers to produce the original byte stream. Many TLS interception tools or network forensics tools do this (for example, after providing keys, Wireshark will show the HTTP protocol decode). So the attacker could feed the decrypted data back into a parser for whatever protocol it is:

If the traffic was an **HTTP** session (most common for web), they will see plaintext HTTP GET/POST requests, headers, and HTML or JSON data responses, etc. They can now parse out URLs, form fields, cookies, API endpoints, etc.

If it was **SMTP/IMAP** for email, they’ll see the email contents in plaintext (the SMTP “DATA” portion or the IMAP fetch results, including email bodies and attachments).

If it was **TLS-secured messaging** (like MQTT for IoT or some chat protocol over TLS), they now read the messages.

For a **VPN inside TLS** (like a TLS-based VPN), they might have to further decode that inner protocol.

In other words, the encryption layer is removed, and whatever application-layer data was being carried is now accessible. The attacker essentially becomes a silent witness to the original communication, with the advantage that they can pause, rewind, or fast-forward as they please (since it’s stored data).

**Extracting Sensitive Data:** Attackers will now comb through the plaintext for **high-value information**. This can often be automated using pattern matching and filtering. Examples of what they look for:

**Login Credentials:** Many protocols involve logins. In HTTP, an attacker might search for HTTP POST requests to /login or any fields named “username” and “password”. They might also extract HTTP **cookies** and session tokens (e.g., Authorization: Bearer <token> headers), which could let them hijack sessions or access accounts. If Basic Authentication was used (Authorization header with base64), that’s immediately apparent. Since the traffic might include entire web pages, they can find credentials either in form submissions or even in the HTML (some pages might stupidly include secrets, or they might find password reset links, etc.).

**Financial Data:** They will scan for things like credit card numbers and bank details. Credit card numbers follow a pattern (16 digits with valid prefixes and a Luhn checksum). Automated regex matching can pull those out easily ([Data Classification Tips: Finding Credit Card Numbers - Varonis](https://www.varonis.com/blog/data-classification-tips-finding-credit-card-numbers)). For example, if someone made a purchase, the attacker could find the card number, expiry, CVV if it passed through TLS (e.g., an API call or page submission). The attacker may maintain a library of regex for various sensitive data: SSNs, bank account numbers, crypto wallet keys, etc. **Data Loss Prevention (DLP)** tools use similar regex patterns to detect sensitive info ([Looking for credit card data and other PII in files](https://security.stackexchange.com/questions/82727/looking-for-credit-card-data-and-other-pii-in-files)), and here the attacker basically uses DLP in reverse – to find juicy data to exploit.

**Personal Communications:** If the TLS session was carrying emails, the attacker can read those emails now. They could collect any personal information, attachments, contacts mentioned, etc. For messaging or social media traffic, they could read private messages. For example, if this was a TLS connection to a cloud API or a chat server carrying user messages, those messages are now in the open. This could include anything from business confidential discussions to medical information.

**System or API keys:** Some communications might include tokens or API keys (for instance, a mobile app might send an API key over TLS). The attacker will be on the lookout for any strings that look like keys or tokens (often a certain length or format, e.g., a JWT token, which can be recognized by its structure). They might also find encryption keys exchanged (though most key exchanges wouldn’t be in plaintext in the data, but sometimes applications do share symmetric keys within TLS).

**Files and Documents:** If files were transmitted (say, an attachment or a downloaded document), the attacker can reconstruct those files from the decrypted stream. They might pull out PDFs, images, or any content and then scan those as well for embedded data.

All these extraction tasks can be automated. The attacker likely has a pipeline:

Take decrypted plaintext of a session.

Identify protocol (maybe via port or by scanning content for telltale signs like HTTP/1.1).

Pass it to a relevant parser or pattern-matcher. HTTP can be parsed to structured form (there are libraries to parse HTTP text into fields). Same for SMTP, etc.

Once structured, apply filters for interesting info.

For example, an automated scanner might output:

“Session X (time, IPs) – HTTP login to example.com – credentials captured: user=alice, pass=Wonderland123”

“Session Y – Credit card number 4111 1111 1111 1111 seen in POST to paymentgateway.com”

“Session Z – Email from [bob@example.com](mailto:bob@example.com) to [charlie@example.net](mailto:charlie@example.net): Subject: Project Plans, attachment Project.pdf (extracted)”

The attacker can then store these findings in a database of **decrypted secrets**. Intelligence agencies likely feed such results into search engines like XKEYSCORE (as reported in Snowden leaks, where analysts could query collected data for email addresses, phone numbers, etc.). In our context, an attacker could categorize decrypted data:

Credentials go into a **credential database** (for later use in intrusions or sold on dark markets).

Credit cards and financial info go to a financial fraud team (or monetized quickly).

Personal info gets indexed by person or topic (for espionage or blackmail).

Confidential documents are stored and perhaps passed to analysts if they belong to targets of interest (e.g., designs from a defense contractor’s traffic).

Communication patterns or metadata might also be gleaned (like who is talking to whom, at what times, which could complement other intel).

It’s worth highlighting how damaging this can be. Data that was encrypted and assumed to be secure can include **passwords, trade secrets, legal documents, medical records, personal photos** – the whole gamut. One reason this HNDL scenario is especially concerning is the *longevity* of certain secrets. Some data loses value quickly (a password can be changed, a credit card can be canceled). But other data has a long shelf life:

Personal emails or messages could be embarrassing or harmful even decades later.

Government secrets or diplomatic communications might still be sensitive many years on (think of intelligence about defense plans or political strategies).

Biometric data, genetic data, etc., once leaked, is permanently sensitive.

This is why experts warn that *“data captured today could be decrypted with ease when Q-Day arrives”* ([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ). The attacker, after decrypting, can exploit the information in whatever way suits their agenda – be it financial gain, identity theft, corporate espionage, or national security espionage.

Finally, once the attacker has decrypted and extracted what they want, they will **store the plaintext (or extracted info)** appropriately. They might not keep all plaintext forever (that’s a lot of data), but they will likely store all the critical extracted pieces in databases that are much smaller. For instance, they might discard the full HTML of a web page but keep the password that was submitted through it. Or they might transcribe a voice-over-IP call (if that was transported over TLS) to text and keep the text if it’s relevant.

In summary, at this stage the attacker is **like an archivist reading previously locked diaries** – but doing so at scale with automation. The once-encrypted TLS 1.3 application data is now available in clear form, and all the secrets it contained are laid bare. The attacker’s goals of the HNDL attack are realized here: the confidentiality of TLS has been compromised, and sensitive information can be exploited directly.

7. Automation & Attack Optimization

Executing a quantum-enabled HNDL attack against potentially **millions of TLS 1.3 sessions** is a Herculean task. Automation, optimization, and smart targeting are crucial for attackers to make the most of their capabilities. In this final phase, we discuss how attackers refine their approach using machine learning (ML), heuristics, and prioritization to maximize the value of the decrypted data:

**Prioritizing High-Value Targets:** Not all encrypted traffic is equally interesting. An attacker will want to focus on sessions that are likely to yield valuable intelligence or financial gain. Early in the pipeline – even before decryption – they can use metadata to flag high-value sessions. For instance, sessions to government domains (.gov or known military IP ranges), major corporate networks, banks and financial services, healthcare providers, etc., are high priority. If the captured metadata includes the Server Name Indication (SNI) or IP address info, they can cross-reference that with known targets. Nation-state attackers likely maintain target lists (e.g., “monitor all traffic involving the defense ministry”, “finance ministry”, or specific dissidents). They might queue those sessions for quantum cracking **first**. Since quantum computing resources might be limited or expensive, this triage ensures the most payoff for the effort. One could imagine an ML system trained on network metadata to predict the “value” of a session – perhaps using features like the server’s identity, the timing (e.g., traffic during business hours of a government office), sizes of data transfer (large transfers might be data-heavy interactions, small might be less interesting, or vice versa if small indicates a login).

**Machine Learning for Traffic Classification:** Even when encrypted, TLS traffic has patterns. Researchers have shown machine learning can classify encrypted traffic by application (VPN vs web vs VoIP, etc.) ([Encrypted Network Traffic Analysis and Classification Utilizing ...](https://pmc.ncbi.nlm.nih.gov/articles/PMC11175201/)). Attackers could use similar techniques on the harvested data to sort it. For example, ML might distinguish a typical web browsing session (multiple short-lived connections to ad servers, CDNs, etc.) from a VPN tunnel (long-lived, constant bitrate) or from an email server connection (regular SMTP patterns). This helps allocate resources – perhaps skip known mundane web surfing but pay attention to a weird persistent TLS connection to an unusual server. However, once decryption keys are available, classification becomes trivial by reading the plaintext protocol. So ML would be more useful in the pre-decryption phase for selection, or post-decryption for extracting insights if volume is huge.

**Matching Client Hellos to Application Data via Heuristics:** In messy real-world data, sometimes packets might be missing or out-of-order in the collection (due to capture issues). Automated heuristics can double-check that the ClientHello and the ensuing data are correctly paired. For example, the system might correlate the ClientHello’s random and the first encrypted handshake message (ServerHello/EncryptedExtensions) by timing and 5-tuple. If there’s any ambiguity (like multiple overlapping sessions on same IP/port – uncommon but could happen with session reuse or NAT), the system uses things like the sequence of TCP ports or unique random values to resolve it. Essentially, this ensures that when a key is recovered, it’s applied to the exact data from that handshake’s connection, not to a wrong one. These checks and balances can be coded in the tooling.

**Field Extraction with AI:** Once the data is decrypted, the sheer volume of plaintext could be enormous – far beyond what human analysts can read. Automation is needed to extract the useful bits (as we discussed with pattern matching). Machine learning can enhance this by understanding context. For example, rather than simple regex for passwords, an NLP (natural language processing) model could be used on chat logs to identify if someone shared contact info or addresses. Or an ML model might classify documents (e.g., “this PDF looks like a financial report” vs “this PDF is a personal letter”). Anomaly detection algorithms might flag things like someone sending out a large database dump. Essentially, after bulk decryption, **big data analytics** steps in. Government attackers likely integrate this into existing intelligence pipelines. For example, the NSA’s XKEYSCORE system reportedly allows querying collected data for selectors like email addresses or keywords – something only possible after decryption. They may also use **analytics to link data**: If they decrypt a username and password on one site, they might automatically try that password on other accounts of the same user (assuming they can find those). That’s more of an active follow-up attack, but it’s enabled by the passive decryption.

**Optimizing Quantum Resource Use:** A unique aspect of HNDL in the quantum age is that quantum computing time will be precious. Attackers won’t waste quantum cycles on low-value targets. So they may use predictive scoring to decide which handshakes to decrypt first or at all. For instance, they might ignore all sessions to Netflix or YouTube (if identified by SNI or IP) because those likely only contain entertainment content, not secrets – unless targeting a specific person’s viewing habits for some reason. On the other hand, any session involving known intelligence targets gets top priority. They might also prioritize by the *type of key exchange*: if some sessions used extremely strong post-quantum algorithms, skip those (no point). Focus on the ones using classical ECC/RSA which they can crack. Government communications using legacy VPNs or older TLS could ironically be easier to break if they used outdated algorithms; an attacker might catalogue those as “easy pickings” for quantum to tackle first.

**Continuous Learning:** Attackers could feed the results of decrypted data back into improving their models. Suppose their system flags certain patterns in encrypted traffic that later (after decryption) turn out to be very fruitful – they can train the system to recognize those patterns earlier. For example, maybe they notice that connections to a certain CDN domain often accompany API calls containing auth tokens to a major cloud provider (because the CDN fronts the API). In the future, just seeing that pattern might cause them to prioritize those sessions. Or if a particular length/time distribution of packets correlates with, say, interactive remote access sessions (like an SSH over TLS scenario), they might target those as they could yield passwords or command logs.

**Case Study – Past Cryptographic Breaks:** We can draw a parallel to how automation helped in previous large-scale crypto-breaking efforts. During WWII, Bletchley Park built automated machines (the Bombes) to help break Enigma daily – an early example of optimizing cryptanalysis workflow. In the 90s, the EFF built “Deep Crack,” an optimized machine, to brute-force 56-bit DES in days. In our context, the “machine” is a combination of quantum computers for the math and classical computers for data processing – both need orchestrating. The attackers will likely automate the entire pipeline: from sniffing to storage to quantum job scheduling to key derivation to plaintext analysis. Each step might have its own optimizations (e.g., scheduling quantum jobs by grouping by curve type to re-use quantum Fourier transform circuits efficiently, etc.).

**Stealth and Efficiency:** An interesting aspect – though not a direct ask in the question but worth noting – is that HNDL can be done entirely passively until the moment of decryption, which could be many years later. This means attack detection is incredibly difficult (no target will know their traffic was harvested). By the time decryption happens, the data is long gone from the source. Attackers thus don’t have to rush; they can optimize leisurely. They might even wait until they have a batch of thousands of similar keys and run them in one go on a quantum computer to amortize overhead. All of these are strategic choices that automation can handle (e.g., queue management, batch processing).

In summary, automation and optimization turn the daunting task of cracking and analyzing vast amounts of TLS 1.3 traffic into a scalable operation. Machine learning assists in focusing efforts where the payoff is highest and in digging out the golden nuggets of intel from mountains of data. Heuristics ensure the right data is connected with the right keys and context. The entire attack becomes an **industrialized pipeline of surveillance**: capture, catalog, crack, and collect insights. What once might require an army of analysts can be done with smart software and enough computing power. This level of automation and sophistication is why experts are concerned that *“some researchers believe HNDL attacks are already happening, likely by nation-states”* ([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ) – the groundwork (mass interception infrastructure) exists, and the only missing piece is the fully capable quantum decryption, which is a matter of time.

**Conclusion:**  
We have traced the full trajectory of a hypothetical Harvest-Now, Decrypt-Later quantum attack on TLS 1.3 – from siphoning encrypted bytes off the wire, to cracking the handshake with quantum algorithms, to finally reading secrets that were meant to stay between the client and server. This process showcases the potential future *breakdown of TLS 1.3’s assurances* under the looming threat of quantum computers. It’s a sobering thought that confidential data protected by robust encryption today could be an open book tomorrow ([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ). Real-world precedents, like past mass-surveillance programs and the eventual cracking of once-strong ciphers (40-bit SSL, 56-bit DES, even 1024-bit RSA in labs), remind us that cryptographic security has a shelf life. The difference with quantum attacks is the **suddenness** – a large quantum leap (pun intended) could render a whole class of algorithms obsolete nearly overnight, exposing years of stored data.

For now, TLS 1.3 remains unbroken by classical means, and deploying perfect forward secrecy was a crucial step to limit damage from key compromises. But the advent of quantum computing threatens to turn the forward-secret model on its head by retroactively nullifying the secrecy of past sessions ([TLS interception - ORG Wiki](https://wiki.openrightsgroup.org/wiki/TLS_interception)) ([Harvest Now, Decrypt Later – Fact or Fiction? | Entrust](https://www.entrust.com/blog/2023/11/harvest-now-decrypt-later-fact-or-fiction)). The defense, not covered in depth here, will be a transition to post-quantum cryptography (PQC) – algorithms believed secure against quantum attacks. Efforts are already underway (e.g., NIST’s PQC standardization, and hybrid TLS key exchanges being tested ()). Yet, the inertia in deploying new cryptography means there’s a window of vulnerability: data being exchanged now with classical algorithms could be in attackers’ storage, *waiting* for that Q-Day to come ([encryption - What is stopping someone from saving encrypted info, and decoding it later? - Cryptography Stack Exchange](https://crypto.stackexchange.com/questions/27728/what-is-stopping-someone-from-saving-encrypted-info-and-decoding-it-later)).

From an attacker’s perspective, as we’ve described, the HNDL approach is a waiting game combined with systematic data mining. As quantum technology matures, what we consider an “impossible” attack today (breaking 256-bit ECC) could become just another high-performance computing task tomorrow. The implications extend beyond just TLS 1.3 – VPNs, encrypted databases, cryptocurrency keys, anything relying on the same math could be at risk.

In closing, the Harvest-Now, Decrypt-Later scenario underscores a fundamental truth in cybersecurity: **today’s “secure” might not be secure forever**. Attackers with foresight and resources operate on long timelines. For researchers, businesses, and policymakers, this means proactive adaptation is key – or else face the consequences when the encrypted troves of yesterday become the open secrets of tomorrow. While our focus here was on the mechanics of the attack (and not on defenses), it’s clear that upgrading our cryptographic arsenal (to quantum-safe algorithms) *before* quantum attackers become operational is critical to thwarting this chilling attack model ([PQC & Crypto-agility: Protect Against Steal & Decrypt Later](https://www.cryptomathic.com/blog/how-to-protect-yourself-against-steal-now-decrypt-later)). The clock is ticking, and both attackers and defenders know it.

----------------------

**References:** (The citations **【】** in the text correspond to the following sources)

([GCHQ taps ALL transatlantic network traffic, says investigative journalist Duncan Campbell](https://www.computing.co.uk/news/2277868/gchq-taps-all-transatlantic-network-traffic-says-investigative-journalist-duncan-campbell)) Computing.co.uk – *GCHQ taps ALL transatlantic network traffic...* (Tempora program description)  
([GCHQ taps ALL transatlantic network traffic, says investigative journalist Duncan Campbell](https://www.computing.co.uk/news/2277868/gchq-taps-all-transatlantic-network-traffic-says-investigative-journalist-duncan-campbell)) Computing.co.uk – Tempora secret feed of fiber-optic traffic  
([Leaked NSA Doc Says It Can Collect And Keep Your Encrypted ...](https://www.forbes.com/sites/andygreenberg/2013/06/20/leaked-nsa-doc-says-it-can-collect-and-keep-your-encrypted-data-as-long-as-it-takes-to-crack-it/)) Forbes/NSA leak – NSA can collect and indefinitely keep encrypted data  
([encryption - What is stopping someone from saving encrypted info, and decoding it later? - Cryptography Stack Exchange](https://crypto.stackexchange.com/questions/27728/what-is-stopping-someone-from-saving-encrypted-info-and-decoding-it-later)) Crypto.SE – Answer: “Nothing stops from saving encrypted info... three-letter agencies build large data centers... waiting for large quantum computer.”  
([A Walkthrough of a TLS 1.3 Handshake](https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art080)) Commandlinefanatic – TLS 1.3 ClientHello showing client random and session ID fields  
([A Walkthrough of a TLS 1.3 Handshake - Command Line Fanatic](https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art080)) wolfSSL blog – TLS Session ID vs Tickets (TLS1.3 uses session ID only for compat with <=1.2)  
([ssl.log — Book of Zeek (git/master)](https://docs.zeek.org/en/master/logs/ssl.html)) Zeek documentation – In TLS1.3, passive observers see server name, cipher, curve, but not certificates (which are encrypted)  
([The Illustrated TLS 1.3 Connection: Every Byte Explained](https://tls13.xargs.org/)) Illustrated TLS13 – Client sends ephemeral public keys in KeyShare extension  
([The Illustrated TLS 1.3 Connection: Every Byte Explained](https://tls13.xargs.org/)) Illustrated TLS13 – KeyShare extension bytes (example for X25519)  
([The Illustrated TLS 1.3 Connection: Every Byte Explained](https://tls13.xargs.org/)) Illustrated TLS13 – ServerHello KeyShare with X25519 public key bytes  
([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ) Packetlabs – Q-Day estimates (>50% chance in 15 years quantum can break RSA-2048 in 24h)  
([Harvest Now, Decrypt Later – Fact or Fiction? | Entrust](https://www.entrust.com/blog/2023/11/harvest-now-decrypt-later-fact-or-fiction)) Entrust blog – Explanation of HNDL: intercept TLS handshake, later use quantum to factor asymmetric keys, retrieve symmetric key, decrypt data  
([PQC & Crypto-agility: Protect Against Steal & Decrypt Later](https://www.cryptomathic.com/blog/how-to-protect-yourself-against-steal-now-decrypt-later)) Cryptomathic – Quantum will solve RSA, DSA, ECDSA (factorization, discrete log), break majority of current asymmetric crypto  
() Wireshark/Sharkfest PDF – “Powerful quantum computers in 15-40 years… essentially all Internet traffic today can be decrypted by these.”  
([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ) Packetlabs – Data captured today could be decrypted with ease when Q-Day arrives via HNDL attacks  
([Q-Day And Harvest-Now-Decrypt-Later (HNDL) Attacks](https://www.packetlabs.net/posts/q-day-and-harvest-now-decrypt-later-attacks/) ) Packetlabs – Some believe HNDL attacks are already happening by nation-states  
([TLS interception - ORG Wiki](https://wiki.openrightsgroup.org/wiki/TLS_interception)) ORG Wiki (TLS interception) – With full access to private keys, captured traffic can be decrypted retrospectively; but not if Diffie-Hellman was used (forward secrecy prevents recreation of session key)  
([TLS 1.2 vs. 1.3—Handshake, Performance, and Other Improvements](https://www.catchpoint.com/http2-vs-http3/tls1-2-vs-1-3)) Catchpoint blog – TLS 1.3 uses forward secrecy; describes ECDH handshake generating unique session key, discarded after session  
([RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3](https://datatracker.ietf.org/doc/html/rfc8446)) RFC8446 (via IETF data tracker) – TLS 1.3 Key Schedule uses HKDF extract/expand with shared secret  
([The TLS 1.3 key schedule. The values of context and label inputs (H](https://www.researchgate.net/figure/The-TLS-13-key-schedule-The-values-of-context-and-label-inputs-H_fig2_353594502)) ResearchGate – Diagram of TLS1.3 traffic key calculation with context/labels (nonce derivation etc.)  
() Sharkfest PDF – Chrome/Firefox enabling hybrid X25519+Kyber (draft) for TLS 1.3; lists “Supported Groups with X25519Kyber768” etc. (indicating hybrid PQC usage)  
([New Cuttlefish malware infects routers to monitor traffic for credentials](https://www.bleepingcomputer.com/news/security/new-cuttlefish-malware-infects-routers-to-monitor-traffic-for-credentials/)) BleepingComputer – *Cuttlefish malware infects routers to monitor data*, creating proxies/VPN tunnels for exfiltration  
([New Cuttlefish malware infects routers to monitor traffic for credentials](https://www.bleepingcomputer.com/news/security/new-cuttlefish-malware-infects-routers-to-monitor-traffic-for-credentials/)) BleepingComputer – Cuttlefish passively sniffs packets for credential markers (usernames, passwords, tokens) in traffic  
([New HiatusRAT router malware covertly spies on victims - Lumen Blog](https://blog.lumen.com/new-hiatusrat-router-malware-covertly-spies-on-victims/)) Lumen Black Lotus – Hiatus malware on routers deploys tcpdump for packet capture on target device  
([New HiatusRAT router malware covertly spies on victims - Lumen Blog](https://blog.lumen.com/new-hiatusrat-router-malware-covertly-spies-on-victims/)) Lumen Black Lotus – Infected router passively sniffs traffic on email/FTP ports and can send it to C2 (Figure: sniff then upload to C2)  
([encryption - What is stopping someone from saving encrypted info, and decoding it later? - Cryptography Stack Exchange](https://crypto.stackexchange.com/questions/27728/what-is-stopping-someone-from-saving-encrypted-info-and-decoding-it-later)) Crypto.SE comment – “The NSA is supposed to do this according to Snowden: save TLS encrypted traffic and wait for more computing power or access to secret key (e.g., with Heartbleed).”  
([TLS interception - ORG Wiki](https://wiki.openrightsgroup.org/wiki/TLS_interception)) ORG Wiki – Even in encrypted TLS, some data like client IP, server IP, and SNI hostname are visible in plaintext  
([TLS 1.2 vs. 1.3—Handshake, Performance, and Other Improvements](https://www.catchpoint.com/http2-vs-http3/tls1-2-vs-1-3)) Catchpoint blog – Describes 0-RTT and its replay risks: attacker could replay client’s early data to server since server can’t initially tell if it’s legit (0-RTT not recommended due to replay issues)  
([The elliptic curve discrete log problem is hard because there is no ...](https://www.reddit.com/r/badmathematics/comments/uaoi7g/the_elliptic_curve_discrete_log_problem_is_hard/)) Reddit /r/cryptography – “A modified version of Shor’s Algorithm can be used to solve the elliptic curve discrete log problem.” (Affirming Shor breaks ECC)  
([PQC & Crypto-agility: Protect Against Steal & Decrypt Later](https://www.cryptomathic.com/blog/how-to-protect-yourself-against-steal-now-decrypt-later)) Cryptomathic – Urges organizations to migrate to PQC *before* attackers start harvesting sensitive data (i.e., proactive defense against HNDL)  
([Encrypted Network Traffic Analysis and Classification Utilizing ...](https://pmc.ncbi.nlm.nih.gov/articles/PMC11175201/)) Research paper (NCBI) – Survey of ML-driven encrypted traffic analysis and classification (showing ML can classify encrypted traffic without decryption)  
([Looking for credit card data and other PII in files](https://security.stackexchange.com/questions/82727/looking-for-credit-card-data-and-other-pii-in-files)) Security.StackExchange – Discussion of using regex to find credit card numbers and PII in files (common DLP regex usage)