Dual-Key Tunnelling Protocol

DKTP Technology Integration Guide

Revision: 1.0
Date: October 11, 2025

1 Introduction and Scope

Dual-Key Tunnelling Protocol (**DKTP**) is a post-quantum, symmetric/asymmetric tunnelling protocol designed to replace legacy PKI-dependent VPNs and TLS tunnels.
It combines two independent sources of entropy; an **asymmetric key-encapsulation mechanism (KEM)** and a **pre-shared symmetric key (PSK),** to derive separate transmit and receive channel keys.
The handshake authenticates both peers using post-quantum signatures and derives two channel keys, providing **perfect forward secrecy** and **post-compromise security**. After each handshake the PSKs are *ratcheted* by hashing them with the new tunnel keys. DKTP operates independently of traditional certificate authorities, making it suitable for **payment networks**, **cloud services, SCADA infrastructure** and **high-value IoT** deployments.

This guide provides practical instructions for integrating DKTP into different domains. It draws on the official specification, executive summary and source-code API.

2 Protocol Overview

2.1 Composite Handshake

1. **Key generation.** Each host maintains a dktp_local_peer_key structure containing:
    - expiration: expiry time (seconds since epoch);
    - config: 48-byte configuration string identifying the KEM, signature scheme, hash function and symmetric cipher;
    - keyid and peerid: unique 16-byte identities for the local device and its peer;
    - pss: the pre-shared secret (PSK);
    - sigkey and verkey: post-quantum signature signing and verification keys.
2. **Dual-entropy key exchange.** During the handshake each side exchanges KEM ciphertexts and signatures. The shared secret derived from the KEM is XORed with the

PSK to produce two secrets: one for the transmit channel and one for the receive channel. Separate transmit and receive keys prevent bidirectional correlation and allow independent ratcheting.

3. **Authenticated encryption.** Once keys are derived, data packets are encrypted with the **RCS** stream cipher in AEAD mode. Packet headers (flags, sequence number, timestamp, payload size) are authenticated using KMAC to ensure replay protection and integrity.
4. **Ratcheting.** After a handshake completes, the protocol hashes the PSK with the new channel keys (via cSHAKE) to update both local and remote PSKs. If DKTP_ASYMMETRIC_RATCHET is enabled, peers may send an *asymmetric ratchet request* to inject a new KEM secret mid-session.

2.2 API Summary

The client.h and dktp.h headers expose the integration API:

Function	***Description***
void dktp_generate_keypair(dktp_remote_peer_key enckey, dktp_local_peer_key* deckey, const uint8_t keyid[DKTP_KEYID_SIZE])*	Generates a key pair. enckey (public) is distributed to peers; deckey (private) is kept secret.
dktp_errors dktp_client_connect_ipv4(dktp_local_peer_key lpk, dktp_remote_peer_key* rpk, const qsc_ipinfo_ipv4_address* address, uint16_t port, void (*send_func)(dktp_connection_state*), void (*receive_callback)(dktp_connection_state*, const uint8_t*, size_t))*	Initiates a duplex connection to a remote host over IPv4. Supply your **local peer key**, the remote peer's **public key**, the server address/port, and callback functions. A similar function exists for IPv6.
dktp_errors dktp_client_listen_ipv4(dktp_local_peer_key lpk, dktp_remote_peer_key* rpk, void (*send_func)(dktp_connection_state*), void (*receive_callback)(dktp_connection_state*, const uint8_t*, size_t))*	Starts a listener on the specified port to accept one inbound connection. A corresponding IPv6 version is available.
bool dktp_send_asymmetric_ratchet_request(dktp_connection_state cns)*	Sends a ratchet request to renegotiate session keys mid-session (requires DKTP_ASYMMETRIC_RATCHET enabled).

dktp_errors dktp_packet_encrypt(dktp_connection_state cns, dktp_network_packet* packetout, const uint8_t* message, size_t msglen)*	Encrypts a message into an output packet using the current transmit key.
dktp_errors dktp_packet_decrypt(dktp_connection_state cns, uint8_t* message, size_t* msglen, const dktp_network_packet* packetin)*	Decrypts and authenticates a received packet using the receive key.
void dktp_local_peer_key_serialize/deserialize(...)	Serializes or restores a dktp_local_peer_key to/from a byte array for storage.

2.3 Choosing Parameter Sets

DKTP supports multiple post-quantum KEM/signature combinations (Kyber/Dilithium, Dilithium/McEliece, McEliece/SPHINCS+). Select the set by defining one of the compile-time macros in dktp.h (e.g., DKTP_CONFIG_DILITHIUM_KYBER). For maximum security choose McEliece/SPHINCS+; for balanced performance choose Dilithium/Kyber.

3 Key Management and Provisioning

1. **Generate server keys:** Use dktp_generate_keypair() at installation time to create a **public encryption key** (dktp_remote_peer_key) and **private signing key** (dktp_local_peer_key) for each service endpoint. Store the private key securely; distribute the public key to client devices.
2. **Assign device identities and PSKs:** For each device (POS terminal, cloud microservice, PLC, IoT endpoint), assign a 16-byte keyid that uniquely identifies the device and a 16-byte peerid for its counterpart. Generate a 64-byte PSK (pss) using a hardware random source. The PSK should be provisioned into both peers and rotated periodically. When the handshake completes, DKTP will ratchet the PSK.
3. **Serialise keys:** Use dktp_local_peer_key_serialize() and dktp_local_peer_key_deserialize() to store keys in non-volatile memory. On start-up, deserialize the keys into memory. Always zeroize keys when decommissioning devices using dktp_local_peer_key_erase().
4. **Set domain strings:** The static DKTP_DOMAIN_IDENTITY_STRING defines a domain/device identity used in cSHAKE customization. For multi-domain deployments (e.g., separate payment and cloud), modify this string at compile time to reflect the domain: e.g., "QRCS:PAYMENT:DKTP1A" for POS networks or "QRCS:SCADA:DKTP1A" for control networks.

4 Integration into Payment Networks

Payment terminals and ATMs require fast, secure tunnels to authorization servers. DKTP's **dual-entropy handshake** eliminates dependency on RSA/ECC certificate infrastructures and completes quickly (two round trips). It provides forward secrecy and constant-time cryptographic operations, reducing vulnerability to side-channel attacks.

4.1 Architecture

- **Client:** Each POS terminal acts as a DKTP client. It holds a dktp_local_peer_key with its PSK and signature keys and knows the server's dktp_remote_peer_key (public encryption key).
- **Server:** The payment processing gateway runs a DKTP listener. It holds its private dktp_local_peer_key and a dktp_remote_peer_key for each registered device (containing the device's public verification key and PSK).
- **Transport:** DKTP runs over TCP or UDP sockets (provided by qsc_socket in the QSC library). The send/receive callbacks interface your network stack.

4.2 Integration Steps

1. **Provisioning:**
    - At device manufacturing or enrolment, call dktp_generate_keypair() on a secure workstation to generate a server key pair (public rpk, private lpk). Copy the public rpk and the device's lpk into the terminal; copy the private lpk (server side) and the device's public key into the payment server database.
    - Generate a 64-byte PSK and store it in both lpk->pss fields.
2. **Implement send/receive callbacks:**
    - The **send callback** signature is void send_func(dktp_connection_state* cns). Inside this callback, call your OS/network API to transmit the bytes from cns->txcpr (DKTP uses qsc_rcs_state to hold cipher state). The send function is triggered by the DKTP library when a packet is ready.
    - The **receive callback** signature is void receive_callback(dktp_connection_state*, const uint8_t* data, size_t len). When new application data is received (after decrypting and authenticating), your callback processes the plaintext (e.g., payment authorization request) and may respond by calling dktp_packet_encrypt() and writing using send_func.

3. **Establish connection:**
    - The POS terminal calls dktp_client_connect_ipv4(lpk, rpk, &server_address, DKTP_CLIENT_PORT, send_func, receive_callback) where server_address is the IP of the payment gateway. This performs the handshake, authenticates the server's signature key and derives dual channel keys.
    - On the server, call dktp_client_listen_ipv4(lpk_server, rpk_device, send_func, receive_callback). The server will block until a connection arrives, perform the handshake and then return a dktp_connection_state.
4. **Transmit transactions:**
    - To send a request, the POS terminal constructs a dktp_network_packet and calls dktp_packet_encrypt(cns, &packetout, message, msglen). It then calls send_func() to write the packet.
    - The server calls dktp_packet_decrypt(cns, plaintext, &len, &packetin) on incoming data. Validate the return value to ensure no tampering has occurred.
5. **Ratcheting and rotation:**
    - At the conclusion of the connection or on a schedule, call dktp_send_asymmetric_ratchet_request(cns) to inject fresh asymmetric entropy. After ratcheting, the PSK is updated.
    - Periodically re-generate and provision new key pairs and PSKs during maintenance windows.

4.3 Operational Considerations

- **Latency:** DKTP's handshake is constant-time and runs in ~2 round trips. Because both transmit and receive channels derive from independent secrets, the cipher state can pre-compute keys, reducing per-packet overhead.
- **Integration with card networks:** DKTP can encapsulate ISO 8583/ISO 20022 messages within its encrypted payloads. Ensure the application layer handles payment message framing.
- **High availability:** Use separate PSKs and key IDs for fail-over servers to prevent key reuse across clusters. When using load balancers, implement sticky sessions so that handshake and ratchet states remain consistent per connection.

5 Integration into Cloud Platforms

Cloud services often span multiple micro-services across data centers. DKTP can secure East–West traffic between services as well as client–server connections.

5.1 Use Cases

- **Secure micro-service RPC:** Replace mutual TLS between micro-services with DKTP to avoid certificate management overhead and provide post-quantum security.
- **Hybrid VPN tunnels:** Create DKTP tunnels between customer networks and cloud gateways for secure remote access. Each tunnel can be bound to a tenant or service account.
- **SaaS API protection:** Wrap API calls within DKTP packets to protect sensitive data in transit.

5.2 Integration Steps

1. **Key distribution service:** Use your existing secret-management system (e.g., HashiCorp Vault, AWS KMS) to generate and store dktp_local_peer_key and dktp_remote_peer_key objects. Provide an API for micro-services to retrieve their keys and PSKs at start-up.
2. **Service Mesh:** For service-mesh architectures (Kubernetes, Istio), implement a sidecar that executes the DKTP handshake on behalf of the service. The sidecar performs dktp_client_connect_ipv4/ipv6 to remote sidecars, passing messages via shared memory or Unix domain sockets.
3. **High concurrency:** DKTP supports multiple concurrent connections. Manage connection states using the functions in connections.h. When a new service instance starts, allocate a dktp_connection_state via dktp_connections_add(), initiate handshake, and store the state for subsequent packet encryption/decryption.
4. **Load balancing:** Because DKTP handshakes embed the peer ID in the header, inbound packets can be routed to the correct connection state. However, ensure that incoming packets from a given peer go to the same backend instance; otherwise, handshake state will not match.
5. **Monitoring and logging:** Use dktp_log_message() and dktp_log_error() to log events. Monitor error codes returned from dktp_packet_encrypt/decrypt() and ratchet functions.

6 Integration into SCADA and Industrial Control Systems

Industrial systems (power grids, manufacturing lines) require deterministic timing and high resilience. Many devices lack hardware cryptography. DKTP offers a lightweight yet high-assurance tunnel that can protect Modbus, DNP3 and proprietary protocols.

6.1 Deployment Architecture

- **Field devices** (RTUs/PLCs) act as DKTP clients. Each holds a dktp_local_peer_key and knows the control server's dktp_remote_peer_key.
- **Control center** runs a DKTP listener per field network. The control center may maintain separate PSKs for each device group (substation, plant area) and may compile DKTP with Dilithium/Kyber for moderate performance.

6.2 Integration Steps

1. **Resource assessment:** Evaluate CPU, RAM and network bandwidth. PQ KEMs (Dilithium/McEliece) require more cycles than AES; ensure the device can complete the handshake within acceptable time (tens of milliseconds on typical ARM microcontrollers). For devices lacking PQ capability, use a proxy gateway to perform DKTP on their behalf.
2. **Key provisioning:** Use offline tools to generate key pairs and PSKs for each device. Embed the PSK in tamper-resistant memory. Because SCADA networks may be isolated, select long key lifetimes (e.g., six months) and schedule maintenance windows for rotation.
3. **Connection management:** Field devices initiate connections using dktp_client_connect_ipv4/ipv6. Control centers call dktp_client_listen_ipv4/ipv6 to accept. The handshake ensures mutual authentication and forward secrecy.
4. **Integration with fieldbus protocols:** After establishing the tunnel, wrap Modbus/DNP3 frames into DKTP packets. On the receiving side, decrypt and feed the plaintext into the existing protocol stack. Consider adjusting timeouts to account for PQ cryptographic processing.
5. **Keep-alive and watchdog:** SCADA links often require continuous liveness monitoring. Implement periodic keep-alive messages using your application layer, or call dktp_send_asymmetric_ratchet_request() regularly to refresh keys and keep the session active.

7 Integration into IoT Devices

DKTP can protect high-value IoT devices such as smart meters, drones and medical equipment. However, PQ cryptography may tax constrained hardware.

7.1 Integration Guidelines

1. **Select parameter sets:** For microcontrollers with limited resources, compile DKTP with the **Dilithium-Kyber** set (DKTP_CONFIG_DILITHIUM_KYBER) to reduce key sizes and handshake time. When security is paramount (e.g., medical devices), use **McEliece-SPHINCS+.**
2. **Memory footprint:** Use -Os compiler optimization and remove unused features (disable DKTP_ASYMMETRIC_RATCHET if frequent ratcheting isn't required). Set DKTP_CONNECTIONS_MAX at compile time to limit the number of concurrent connections and reduce memory usage.
3. **Key storage:** Store the PSK and key identifiers in secure flash or a trusted platform module (TPM). Zeroize memory after handshake. Consider deriving PSKs from device secrets using HKDF at start-up to avoid storing long secrets.
4. **Connection initiation:** IoT devices typically act as clients. When connecting to cloud or local hubs, call dktp_client_connect_ipv6() using the IPv6 address of the gateway. Implement event-driven send/receive callbacks integrated with the device's network stack (e.g., lwIP). Use non-blocking sockets to avoid blocking the control loop.
5. **Firmware updates:** To update device firmware via DKTP tunnels, ensure the download service uses dktp_packet_encrypt() to transmit signed and encrypted update chunks. After verifying the update signature, the device can apply the patch securely.

8 Security Best Practices

- **Protect private keys and PSKs:** Always store dktp_local_peer_key.sigkey and pss in secure memory. On systems with secure enclaves, isolate these secrets using hardware keys. Use constant-time operations when comparing or copying secrets to prevent side-channel leaks.
- **Identity management:** Use unique keyid and peerid values per connection. Avoid reusing the same PSK across multiple devices or services.
- **Certificate independence:** DKTP does not require external CA certificates; however, if you integrate with legacy systems, ensure certificates are validated separately before entering DKTP tunnels.
- **Replay and downgrade protection:** DKTP authenticates packet headers (flag, sequence, timestamp, payload size) using KMAC. Always check return codes from dktp_packet_decrypt() and abort the connection on failure.
- **Ratcheting schedule:** Even with PSK ratcheting, plan regular key rotations. Use asymmetric ratchet requests during long-lived connections to inject fresh entropy.

9 Conclusion

The Dual-Key Tunnelling Protocol offers a high-security, certificate-independent alternative to classical VPN and TLS tunnels. By combining post-quantum KEMs with pre-shared symmetric keys, DKTP derives two independent channels providing forward secrecy and strong replay protection. Its modular API, as demonstrated in the client.h and dktp.h headers, enables integration into payment systems, cloud platforms, SCADA networks and high-value IoT devices.

Adopting DKTP requires careful key management and provisioning, selection of parameter sets appropriate to your hardware, and implementation of send/receive callbacks tied to your network stack. Following the integration guidelines outlined here will allow you to deploy DKTP securely and effectively across diverse infrastructures.