Post Quantum Shell

PQS Technology Integration Guide

Revision: 1.0
Date: October 14, 2025

1 Introduction and Scope

The **Post Quantum Shell (PQS)** is a high-security, post-quantum communication protocol designed to replace classical Secure Shell (SSH) for remote server access. Developed by QRCS, PQS employs a **one-way trust model** in which the *client trusts the server*, and uses a **Simplex key exchange** to create a 256-bit post-quantum secure session. The protocol leverages modern post-quantum cryptographic primitives; **Kyber** or **McEliece** for key encapsulation, **Dilithium** or **SPHINCS+** for signatures, and the **RCS** (Rijndael Cryptographic Stream) cipher with **SHAKE/KMAC**, to provide confidentiality, integrity and forward secrecy even in the presence of quantum adversaries. Unlike SSH, which maintains long-lived host keys, PQS creates unique, per-session keys and stores less than 4 kB of state per client, enabling a single server to handle hundreds of thousands of concurrent sessions.

PQS was designed for industries requiring long-term security and regulatory compliance such as **financial technology**, **government communications**, **healthcare**, **critical infrastructure** and **cloud**. It can also secure registration events, hub-and-spoke networks, commodity trading and electronic currency exchange. This guide provides practical instructions for integrating PQS into payment networks, cloud platforms, SCADA systems and IoT devices. It draws on the official specification and executive summary as well as the PQS source code to outline the API, key management and deployment patterns.

2 Protocol Overview

2.1 One-Way Trust and Simplex Key Exchange

PQS follows a **one-way trust model**: the client authenticates the server but not vice versa. A client is provisioned with the server's *public signature verification key* and a 16-byte **Key Identity (kid)**. The client initiates a connection by sending its kid and a **configuration string** describing the cryptographic suite (signature scheme, encapsulation algorithm, hash family and symmetric cipher). Both peers compute a **session cookie**; a 256-bit hash of the configuration string, key identity and server verification key, to bind the exchange to a particular session. The handshake proceeds in four stages:

1. **Connect request:** The client sends {*kid*, *cfg*} to the server. It computes *sch* ← H(*cfg* || *kid* || *pvk*) as its session cookie.
2. **Connect response:** The server checks that it owns the requested key, verifies the protocol string, computes its own sch, generates a fresh key encapsulation pair (*pk*, *sk*), hashes *pk* and the response header, signs this hash with its signature key, and sends back {*spkh*, *pk*}, the signed hash and the public KEM key. Clients verify the signature before continuing.
3. **Exchange request:** The client uses the server's pk to encapsulate a shared secret *sec*, derives two session keys (*k1*,*k2*) and two nonces (*n1*,*n2*) from *sec*||*sch* via a Keccak-based key derivation function (KDF), initializes the transmit cipher with (*k1*,*n1*) and receive cipher with (*k2*,*n2*) and sends the encapsulated secret *cpt* to the server.
4. **Exchange response:** The server decapsulates the shared secret, derives (*k1*,*k2*,*n1*,*n2*) in the same way, initializes its receive and transmit ciphers (using the reverse key order), and responds with an exchange-response flag to signal that the encrypted tunnel is ready. Both peers then mark the connection as established.

After establishment, the session keys are used to encrypt and decrypt messages using RCS in **AEAD** mode. Each message carries a small header (21 bytes) containing a *packet flag*, *sequence number*, *payload length* and *timestamp*, all of which are authenticated via KMAC. PQS uses a keep-alive mechanism and sequence numbers to prevent replay and stale sessions.

2.2 Cryptographic Primitives

PQS uses only post-quantum cryptographic primitives:

- **Key encapsulation: Kyber** (IND-CCA secure lattice scheme) or **McEliece** (code-based KEM) to establish a shared secret.
- **Signatures: Dilithium** (lattice based) or **SPHINCS+** (stateless hash-based) to authenticate the server's KEM key.
- **Symmetric cipher: RCS** (Rijndael Cryptographic Stream), a wide-block adaptation of AES with 256-bit blocks and more rounds; integrated with KMAC for authenticated encryption and uses cSHAKE-based key expansion.
- **Hash and KDF: SHA-3**, **SHAKE** and **KMAC** for hashing, key derivation and message authentication. The KDF produces two independent keys and nonces from the shared secret and session cookie.

2.3 State Structures

PQS defines several state structures that developers must manage:

- **Client key (pqs_client_verification_key):** Contains the server's public signature verification key, the protocol configuration string and a key identity. Fields include an expiration timestamp, a 320-bit configuration string, a 128-bit key ID and a variable-length verification key. Clients store this structure to authenticate the server during each connection.
- **Server key (pqs_server_signature_key):** Private counterpart of the client key. It contains the server's signing key in addition to the fields present in pqs_client_verification_key.
- **Connection state (pqs_connection_state):** Stores cipher instances, sequence counters, connection ID, key exchange flags, ratchet key, a 256-bit session token and the current packet flag. Each active connection has its own state object.
- **Key exchange state (pqs_kex_client_state / pqs_kex_server_state):** Temporary structures used during the Simplex exchange to hold key identities, session tokens, signature keys, shared secrets and expiration times. These are freed after the exchange completes.
- **Network packet (pqs_network_packet):** Holds a packet flag, sequence number, message length, timestamp, and pointers to the message and MAC tag. Packet headers are exactly 21 bytes.

These structures are defined in pqs.h and kex.h and are manipulated by the API functions described below.

2.4 Packet Serialization and AEAD

PQS uses RCS in an AEAD construction. To **encrypt** a message, the sender increments the transmit sequence number, creates a header with the flag, sequence number, and message length, attaches the current UTC time, sets the associated data of the RCS cipher to the serialized header, encrypts the plaintext, computes a KMAC tag over the header and ciphertext, and appends the tag. **Decryption** performs the reverse: verify the timestamp is within a tolerance, deserialize the header, verify the MAC, then decrypt the ciphertext. If verification fails, the connection is terminated.

3 API Summary

PQS is implemented as a C library accompanied by Doxygen documentation. The API separates **key management, client**, **server** and **packet** operations. Function prototypes below are simplified; consult pqs.h, client.h, server.h and kex.h for full signatures.

3.1 Key Management

Function	*Purpose*
void pqs_generate_keypair(pqs_client_verification_key pubkey, pqs_server_signature_key* prikey, const uint8_t kid[PQS_KEYID_SIZE])*	Generate a new asymmetric signature key pair (Dilithium/Sphincs+). The public fields (pubkey) contain the verification key, configuration string and key ID; the private fields (prikey) include the signing key. The expiration timestamp is set automatically.
void pqs_public_key_encode(char enck[PQS_PUBKEY_STRING_SIZE], const pqs_client_verification_key pubkey)*	Serialize a client verification key into a printable string containing the configuration, key ID, expiration time and base64-encoded verification key. Use this to distribute the server's public key to clients.
bool pqs_public_key_decode(pqs_client_verification_key pubk, const char enck[PQS_PUBKEY_STRING_SIZE])*	Parse an encoded public key string and populate a client verification key structure. Returns true if successful.

3.2 Client API

Function	*Purpose*
pqs_errors pqs_client_connect_ipv4(const pqs_client_verification_key pubk, const qsc_ipinfo_ipv4_address* address, uint16_t port, void (*send_func)(pqs_connection_state*), void (*receive_callback)(pqs_connection_state*, const uint8_t*, size_t))*	Connect to a PQS server over IPv4, perform the Simplex key exchange and, on success, start a receive loop on a new thread and execute the caller's send loop on the main thread. Returns an error code on failure.
pqs_errors pqs_client_connect_ipv6(...)	Same as above but for IPv6 addresses.
void pqs_connection_close(pqs_connection_state cns, pqs_errors err, bool notify)*	Close the connection. Optionally send a disconnect or error packet before terminating.

void pqs_connection_state_dispose(pqs_connection_state cns)*	Erase and free the connection state.
pqs_errors pqs_packet_encrypt(pqs_connection_state cns, pqs_network_packet* packetout, const uint8_t* message, size_t msglen)*	Encrypt a plaintext buffer and populate a packet structure. Increments the transmit sequence number and attaches a MAC tag.
pqs_errors pqs_packet_decrypt(pqs_connection_state cns, uint8_t* message, size_t* msglen, const pqs_network_packet* packetin)*	Decrypt an incoming packet after verifying the header and MAC. Copies the plaintext into the provided buffer.
void pqs_packet_header_create(pqs_network_packet packetout, pqs_flags flag, uint64_t sequence, uint32_t msglen)*	Initialize a packet header with the given flag, sequence number and message length.
void pqs_packet_header_serialize(const pqs_network_packet packet, uint8_t* header) / void pqs_packet_header_deserialize(const uint8_t* header, pqs_network_packet* packet)*	Convert between packet structures and raw byte arrays for network transmission.
bool pqs_packet_time_validate(const pqs_network_packet packet)*	Check that a packet's timestamp falls within the allowed time window.

3.3 Server API

Function	***Purpose***
pqs_errors pqs_server_start_ipv4(qsc_socket source, const pqs_server_signature_key* prikey, void (*receive_callback)(pqs_connection_state*, const uint8_t*, size_t), void (*disconnect_callback)(pqs_connection_state*))*	Start a multi-threaded PQS server listening on an IPv4 socket. Accepts new clients, performs the server side of the key exchange and spawns threads to handle receive loops.
pqs_errors pqs_server_start_ipv6(...)	Same as above but for IPv6.
void pqs_server_pause(void) / void pqs_server_resume(void)	Temporarily suspend or resume accepting new connections (maintenance mode).

void pqs_server_quit(qsc_socket source)*	Shut down the server and close all active connections.

3.4 Enumerations and Error Handling

Several enumerations help interpret packet flags and error codes. Relevant values include:

- **pqs_flags** Packet types: pqs_flag_connection_request, pqs_flag_connection_response, pqs_flag_encrypted_message, pqs_flag_connection_terminate and various error flags.
- **pqs_errors** Function return codes: pqs_error_none, pqs_error_connection_failure, pqs_error_exchange_failure, pqs_error_decryption_failure, etc.
- **pqs_messages** Logging categories used by the built-in logger.

Refer to pqs.h for complete definitions. Use pqs_error_to_string() or pqs_error_description() to convert codes to human-readable messages.

4 Key Management and Provisioning

4.1 Generating and Distributing Keys

1. **Create a key identity.** Each PQS server must be associated with a 16-byte kid. Choose a random identifier or derive it from the server's domain. This value helps clients locate the correct public key.
2. **Generate a signature key pair.** Use pqs_generate_keypair() to create a new pqs_server_signature_key and the corresponding pqs_client_verification_key. The function sets the expiration timestamp (default one year) and copies the configuration string and key identity into the structures.
3. **Publish the public key.** Serialize the client verification key with pqs_public_key_encode() and distribute the encoded string to clients through secure channels (e.g., via PKI, QR codes, or provisioning servers). Protect the private signing key in an HSM; never expose it.
4. **Key rotation.** Before the expiration date, generate a new key pair and update the client list. Clients should verify the expiration timestamp when connecting and request a re-authentication session if the key has expired.

4.2 Selecting Cryptographic Parameters

The configuration string defines the cryptographic primitives used by PQS. Currently, the following combinations are supported:

Signature	Encapsulation	Hash	Symmetric cipher
Dilithium-S1, S3, S5	Kyber- S1, S3, S5	SHA3-256	RCS-256
Dilithium- S1, S3, S5	McEliece- S1, S3, S5	SHA3-256	RCS-256
SPINCS+- S1, S3, S5	McEliece- S1, S3, S5	SHA3-256	RCS-256

Asymmetric cipher and signature schemes are rated on the NIST PQC performance levels S1, S3, and S5, corresponding to 128, 192 and 256 bits of security. These values are set in the QSC library where the cipher or signature scheme version are set in the qsccommon.h macros. Dilithium, Kyber and SPHINCS+ have been updated to the latest FIPS standardized versions in QSC. For most deployments, the Dilithium-S5 and kyber-S5 set provides a good balance between performance and security.

4.3 Client Verification Key Handling

Clients must store the server's verification key securely. At start-up, decode the key string with pqs_public_key_decode() and check:

- **Protocol compatibility:** Ensure the client library supports the same configuration string; if not, abort the connection.
- **Expiration:** Compare the stored expiration timestamp with the current UTC time; if expired, request a fresh key.
- **Key identity:** Use the kid to match the key with the target server.

5 Implementation Steps

5.1 Client Integration

1. **Decode the server's verification key.** Call pqs_public_key_decode() on the encoded key string to obtain a pqs_client_verification_key structure. Validate the protocol string and expiration.
2. **Resolve the server address.** Populate a qsc_ipinfo_ipv4_address or qsc_ipinfo_ipv6_address structure with the server's IP address and port (default PQS_SERVER_PORT).
3. **Define callback functions.** Implement a send_func(pqs_connection_state* cns) that sends application data over the secure tunnel. Inside this function:

- Create a pqs_network_packet and call pqs_packet_encrypt(cns, &pkt, message, msglen) to encrypt outgoing data.
- Convert the packet to a byte stream with pqs_packet_to_stream(&pkt, buffer), then send it over the underlying socket.
- Update application logic based on responses.
  Implement a receive_callback(pqs_connection_state* cns, const uint8_t* message, size_t msglen) to handle decrypted messages from the server. This callback runs on a separate thread created by the PQS library.

4. **Connect.** Invoke pqs_client_connect_ipv4() or _ipv6(), passing pointers to the verification key, address, port, send_func and receive_callback. The function performs the Simplex handshake and, on success, begins the receive loop in a new thread and calls your send loop on the current thread.
5. **Handle errors and closure.** If any function returns an error code (pqs_error_*), close the connection with pqs_connection_close(), free the connection state with pqs_connection_state_dispose(), log the error with pqs_log_error() or pqs_log_message(), and attempt reconnection if appropriate.

5.2 Server Integration

1. **Generate or load server keys.** Use pqs_generate_keypair() at installation time to create a pqs_server_signature_key. Alternatively, load a previously generated key from secure storage.
2. **Create a listener socket.** Initialize a qsc_socket for IPv4 or IPv6. Bind it to PQS_SERVER_PORT (default 3119) and start listening.
3. **Provide callback functions.** Implement a receive_callback(pqs_connection_state* cns, const uint8_t* message, size_t msglen) to process incoming messages. Use pqs_packet_decrypt() to decrypt the message and verify the MAC. Implement a disconnect_callback(pqs_connection_state* cns) to clean up state when a client disconnects.
4. **Start the server.** Call pqs_server_start_ipv4() or _ipv6(), passing the listener socket, server key, and callback pointers. The server accepts connections, performs the server side of the key exchange internally using pqs_kex_server_key_exchange(), and spawns threads to handle client receive loops. For each connected client, the receive_callback is invoked whenever a decrypted message is available.

5. **Pause, resume or quit.** Use pqs_server_pause() to temporarily stop accepting new connections during maintenance and pqs_server_resume() to restart. When shutting down, call pqs_server_quit() to close all sockets and free resources.

5.3 Packet Processing

After the handshake, both client and server send and receive packets using the following pattern:

1. **Message preparation:** Fill a pqs_network_packet with your plaintext message and set the appropriate flag (e.g., pqs_flag_encrypted_message). Call pqs_packet_encrypt() to produce a ciphertext and MAC tag. Use pqs_packet_header_serialize() followed by sending the raw bytes over the socket.
2. **Reception:** Read exactly PQS_HEADER_SIZE bytes from the socket to obtain the header. Deserialize with pqs_packet_header_deserialize(), allocate a buffer large enough for the message and MAC, receive the remaining bytes, then call pqs_packet_decrypt() to obtain the plaintext. Validate the timestamp with pqs_packet_time_validate() and discard the message if it is out of the acceptable window.
3. **Sequence and state update:** Both peers maintain separate send and receive sequence counters in the connection state. The library updates these automatically. If a packet flag indicates an error or termination, gracefully close the connection.

6 Integration into Payment Networks

PQS is not a transaction-level encryption protocol like HKDS; rather, it is a **remote shell** that can replace SSH for managing payment infrastructure. In payment networks, PQS enables secure, quantum-resistant administration of ATMs, POS terminals and cardholder data environments.

6.1 Use Cases

- **Remote maintenance:** Securely access and update software on ATMs and POS terminals over untrusted networks. PQS's one-way trust model ensures that devices only connect to authorized servers, mitigating the risk of rogue management servers.
- **Key custodian operations:** Rotate BDKs or STKs in payment processors using PQS tunnels to HSMs and management servers. Because PQS is resistant to quantum attacks, it preserves the confidentiality of cryptographic keys during long-term storage and transfer.

- **Regulatory compliance:** PQS sessions meet PCI DSS requirements for strong authentication and encryption, and support logging via the built-in logging functions for audit trails.

6.2 Deployment Considerations

1. **Provisioning devices:** Embed the server's verification key into terminal firmware during manufacturing or registration. Alternatively, distribute the key via HKDS or SATP channels during initial enrolment.
2. **Network architecture:** Place a PQS server behind a firewall in the payment processor's data center. Devices connect outward over **port 3119** or a custom port. To limit inbound attack surface, restrict connections to whitelisted IPs or use SATP as a transport layer.
3. **Session management:** Use PQS's small memory footprint (<4 kB per client) to support thousands of concurrent maintenance sessions. Configure keep-alive intervals and timeouts appropriate for intermittent connectivity in retail settings.
4. **Command interpreter:** Build a command layer on top of PQS using the pqs_client_commands enumeration (see interpreter.h in the source). Map commands such as update-firmware, rotate-key or status to functions executed on the server side.

7 Integration into Cloud Platforms

PQS excels as a remote administration channel for cloud servers and micro-services. Because it eliminates long-term host keys, it reduces the risk of key compromise and simplifies certificate management.

7.1 Use Cases

- **Secure shell replacement:** Replace OpenSSH/SSH with PQS in bastion hosts or jump boxes. Clients store the server's verification key and connect on demand. Because each session uses fresh keys, compromise of one session does not affect others.
- **DevOps automation:** Use PQS in deployment pipelines to execute remote commands, rotate secrets or patch systems. Integrate the PQS client library into provisioning tools such as Ansible or Terraform.
- **SaaS control planes:** Administrators can log into customer environments through PQS tunnels, ensuring that support and maintenance communications remain quantum-safe.

7.2 Deployment Considerations

1. **Key management:** Maintain the server's signature key in an HSM or key vault. Rotate keys regularly and update client verification keys using a configuration management system.
2. **Scalability:** Because the PQS server is multi-threaded and uses less than 4 kB of state per client, it can support hundreds of thousands of concurrent sessions. Deploy behind a load balancer and replicate the server process across nodes; each instance can derive the same session keys from the private key.
3. **API integration:** Expose PQS client functions as a library or CLI. For example, implement a wrapper that reads commands from standard input, encrypts them with pqs_packet_encrypt() and writes decrypted responses to standard output. For micro-services, integrate PQS calls into the application code or sidecar containers.
4. **Logging and auditing:** Use pqs_log_message() and pqs_log_error() to record session events and errors. Collect logs centrally for audit compliance.

8 Integration into SCADA Systems and Industrial Control

8.1 Use Cases

- **Remote diagnostics:** Connect securely to programmable logic controllers (PLCs), remote terminal units (RTUs) and sensors for diagnostics and firmware updates. PQS's small state and minimal computational overhead make it suitable for embedded controllers.
- **Command-and-control:** Issue control commands to field devices over untrusted networks. The authenticated encryption ensures commands cannot be modified or replayed.

8.2 Deployment Considerations

1. **Client implementation:** Port the PQS client library to the target embedded platform. Ensure that the underlying socket and threading abstractions are available; if not, adapt the library to your RTOS.
2. **Key provisioning:** Load the server's verification key into devices during manufacturing. The devices will not hold any private keys; they simply verify the server and derive session keys on the fly.
3. **Network resilience:** Configure keep-alive timeouts appropriate for intermittent links. Use the pqs_packet_time_validate() function to discard stale packets and tear down idle sessions.

4. **Operational safety:** Combine PQS with a control-layer protocol (e.g., Modbus, DNP3) inside the encrypted tunnel. Validate commands at the application layer before acting on them.

9 Integration into IoT Devices

9.1 Use Cases

- **Secure firmware updates:** Deliver firmware images to IoT devices over PQS tunnels, ensuring integrity and confidentiality. The server's signature key authenticates the update source.
- **Device configuration:** Remote configuration and parameter tuning for smart sensors and home automation devices.
- **Telemetry uplink:** Securely transmit sensor data back to the cloud. Use pqs_packet_encrypt() to wrap telemetry messages and optionally include metadata (e.g., device ID, timestamp) in the MAC's associated data field.

9.2 Deployment Considerations

1. **Resource constraints:** PQS requires about 4 kB of RAM per connection on the server side; the client requires less memory but must allocate buffers for headers and messages. Optimize buffer sizes according to PQS_NETWORK_BUFFER_SIZE and limit simultaneous sessions on extremely resource-constrained devices.
2. **Network addressing:** For devices behind NATs or firewalls, support outbound-only connections using the client API. Use IPv6 where available to simplify addressing and avoid port exhaustion.
3. **Batch operations:** If devices connect intermittently, queue updates and send them once a PQS session is established. The server can hold pending messages and deliver them when the device connects.
4. **Interoperability:** Integrate PQS with other QRCS protocols like **HKDS** for symmetric key derivation or **SATP** for secure tunnels. For example, use SATP to connect devices to a proxy and run PQS over the established channel.

10 Security and Operational Best Practices

- **Verify server keys:** Always check the expiration and configuration string of the server's verification key before initiating a connection. Reject expired or mismatched keys.

- **Enforce time windows:** Use pqs_packet_time_validate() to ensure packets have recent timestamps. Discard packets outside the PQS_PACKET_TIME_THRESHOLD and close the connection to prevent replay attacks.
- **Log and monitor:** Enable logging via pqs_log_message() and pqs_log_error() and send logs to a central monitoring system for intrusion detection and compliance.
- **Limit privileges:** The PQS server runs as a privileged process because it holds the private signing key. Restrict access to this process and use OS sandboxing or containerization.
- **Use strong random:** When generating keys or nonces externally, supply high-entropy random data. PQS uses SHAKE internally, but external randomness is still required for key generation and network operations.
- **Rotate keys regularly:** Replace the server's signature key before expiration. Remove support for old protocol strings to reduce downgrade attacks.
- **Secure transport:** Although PQS provides end-to-end encryption, consider layering it over SATP or TLS for additional protection in highly regulated environments.

11 Conclusion

PQS is a lightweight yet powerful remote shell protocol that brings post-quantum security and forward secrecy to remote administration tasks. By combining a one-way trust model with modern post-quantum primitives, PQS eliminates the need for long-term host keys and scales to hundreds of thousands of concurrent sessions with minimal memory. The API provides clear abstractions for key generation, client and server operations, packet handling and error management. Integrating PQS into payment networks, cloud platforms, SCADA systems and IoT devices enables secure and compliant remote management today while protecting against future quantum adversaries.