Hierarchical Key Derivation System

HKDS Technology Integration Guide

Revision: 1.0
Date: October 12, 2025

1 Introduction and Scope

The **Hierarchical Key Derivation System (HKDS)** is a next-generation, post-quantum key management protocol. It was designed to replace legacy *Derived Unique Key Per Transaction* (DUKPT) schemes used in payment terminals with a faster, infinitely scalable mechanism that eliminates the need for state retention or certificate hierarchies. HKDS combines two long-term master secrets; a **Base Derivation Key (BDK)** and a **Secret Token Key (STK)**, to derive per-device *Embedded Device Keys (EDKs)* and per-transaction *Transaction Key Caches (TKCs)*. The protocol relies exclusively on NIST-approved primitives such as **SHAKE** extendable-output functions and **KMAC** for message authentication, providing quantum-safe security up to 512 bits.

Compared with DUKPT, HKDS delivers dramatic improvements: it uses constant-time operations, produces unique keys for every message without storing state on the server, scales to millions of devices, and supports longer key sizes (128/256/512 bits). The dual-key construction allows token keys to be rotated without re-provisioning devices, yielding forward secrecy and predictive resistance. HKDS is therefore suitable for **payment networks, cloud services**, **SCADA systems**, and **embedded/IoT devices** where high throughput and quantum-safe security are paramount.

This guide provides detailed instructions for integrating HKDS into various environments. It draws on the official specification and executive summary as well as the HKDS source code to outline the API, key management, and domain-specific deployment patterns.

2 Protocol Overview

2.1 Hierarchical Key Derivation

At its core, HKDS uses a hierarchical key structure:

1. **Master Key Set (MDK):** A tuple (BDK, STK, KID) provisioned into the transaction server. The **Base Derivation Key (BDK)** and **Secret Token Key (STK)** are 32-byte secrets. **KID** is a 4-byte identifier that distinguishes different master key sets. The server uses these keys to generate tokens and reconstruct transaction keys.

2. **Key Serial Number (KSN):** A 16-byte identifier assigned to each client. It contains the device identity (**DID**, 12 bytes) and a 4-byte token counter. The KSN increments on every token request to ensure uniqueness.
3. **Embedded Device Key (EDK):** A 32-byte secret derived on the server by concatenating BDK with the client's DID and hashing with SHAKE; it is then provisioned into the device. The EDK remains fixed for the life of the device.
4. **Transaction Key Cache (TKC):** A pseudo-random stream of keys derived on the client by concatenating the decrypted secret token with the EDK and feeding it into SHAKE. The output is segmented into 16-byte transaction keys. Each key is consumed once and then erased to guarantee forward secrecy.
5. **Token Exchange:** When a device exhausts its TKC, it sends a **token request** (packet type packet_token_request) containing its KSN. The server encrypts a new secret token with the STK and sends back an encrypted token (ETOK) with an authentication tag. The client decrypts the token using its EDK after verifying the KMAC tag and regenerates its TKC.

This hierarchy decouples the life cycle of embedded keys from token keys. The EDK never changes, while the STK can be periodically rotated on the server without re-provisioning devices. Forward secrecy and predictive resistance arise because the server cannot predict future transaction keys without the token counter in the KSN.

2.2 Cryptographic Primitives

HKDS relies entirely on SHA-3–derived primitives:

- **SHAKE-128/256/512:** Extendable-output functions used as pseudo-random functions (PRFs) for key derivation. The choice of security level is encoded in the protocol field of the message header, with defined values protocol_shake_128, protocol_shake_256 and protocol_shake_512. SHAKE-256 is the default.
- **KMAC:** A Keccak-based MAC used to authenticate encrypted tokens and messages. MAC tags are 16 bytes (HKDS_TAG_SIZE).

2.3 Packet Structure

All HKDS packets begin with a 4-byte header (hkds_packet_header) containing a **flag** (packet type), **protocol ID, sequence number**, and **length**. For authenticated messages, the ciphertext is followed by a 16-byte MAC tag. Standard packet types include packet_token_request, packet_token_response, packet_message_request and packet_message_response. Administrative and error packet types also exist.

3 API Summary

The HKDS library exposes separate APIs for clients and servers. Vectorized (x8 and x64) variants of many functions exist for high-throughput processing; this guide focuses on the scalar API.

3.1 Server API

Function	***Purpose***
*void hkds_server_generate_mdk(bool (*rng_generate)(uint8_t*, size_t), hkds_master_key* mdk, const uint8_t* kid)*	Generate a new master key set by invoking a caller-supplied random generator to fill the BDK and STK. The KID identifies the key set.
void hkds_server_generate_edk(const uint8_t bdk, const uint8_t* did, uint8_t* edk)*	Derive an Embedded Device Key by hashing the concatenation of the Base Derivation Key and the device identity. Provision this key into the device during manufacturing or enrolment.
void hkds_server_initialize_state(hkds_server_state state, hkds_master_key* mdk, const uint8_t* ksn)*	Initialize the server state by copying the client's KSN, attaching the master key set, and resetting token counters.
void hkds_server_encrypt_token(hkds_server_state state, uint8_t* etok)*	Encrypt the next secret token using the STK and a custom token string; append a MAC tag. This function is used when responding to a token request.
void hkds_server_decrypt_message(hkds_server_state state, const uint8_t* ciphertext, uint8_t* plaintext)*	Decrypt a client message by generating the appropriate transaction key and XORing it with the ciphertext.

bool hkds_server_decrypt_verify_message(hkds_server_state state, const uint8_t* ciphertext, const uint8_t* data, size_t datalen, uint8_t* plaintext)*	Verify a message's MAC and decrypt it. If verification fails, the plaintext buffer is zeroed.

3.2 Client API

Function	Purpose
void hkds_client_initialize_state(hkds_client_state* state, const uint8_t* edk, const uint8_t* did)	Initialize the client state with its EDK and device identity. The KSN is constructed from the DID and a zeroed token counter; the TKC is marked empty.
bool hkds_client_decrypt_token(hkds_client_state* state, const uint8_t* etok, uint8_t* token)	Decrypt an encrypted token received from the server. The client derives a token customization string (CTOK), verifies the MAC (TMS) using KMAC, and if valid, uses SHAKE to decrypt the token.
void hkds_client_generate_cache(hkds_client_state* state, const uint8_t* token)	Generate a new Transaction Key Cache by concatenating the token with the EDK, running SHAKE and splitting the output into 16-byte keys.
bool hkds_client_encrypt_message(hkds_client_state* state, const uint8_t* plaintext, uint8_t* ciphertext)	Encrypt a plaintext message by XORing it with a transaction key drawn from the TKC. The key is then erased from the cache.

Function	Purpose
bool hkds_client_encrypt_authenticate_message(hkds_client_state* state, const uint8_t* plaintext, const uint8_t* data, size_t datalen, uint8_t* ciphertext)	Encrypt a message and append a 16-byte KMAC tag that authenticates both the ciphertext and optional additional data. Returns false if the cache is empty.

3.3 Configuration Constants

The header hkds_config.h defines sizes and identifiers used by HKDS. Important constants include:

Name	Value	Description
HKDS_BDK_SIZE	32 bytes	Size of Base Derivation Key (BDK).
HKDS_STK_SIZE	32 bytes	Size of Secret Token Key (STK).
HKDS_EDK_SIZE	32 bytes	Size of Embedded Device Key (EDK).
HKDS_KSN_SIZE	16 bytes	Size of the Key Serial Number, containing DID (12 bytes) and a 4-byte counter.
HKDS_ETOK_SIZE	48 bytes	Size of an encrypted token (including MAC tag).
HKDS_MESSAGE_SIZE	16 bytes	Size of message blocks.
HKDS_TAG_SIZE	16 bytes	Size of KMAC authentication tag appended to messages and tokens.
HKDS_PROTOCOL_TYPE	protocol_shake_256 (default)	Indicates the SHAKE variant; other values include protocol_shake_128 and protocol_shake_512.

4 Key Management and Provisioning

Effective integration of HKDS begins with secure key provisioning.

1. **Generate the master key set.** Use hkds_server_generate_mdk() to create BDK and STK values. Provide a high-quality random generator (e.g., from a hardware RNG or FIPS 140-2/3 compliant module) to fill the key materials. Assign a unique 4-byte KID for each deployment zone (e.g., per merchant or per data center).
2. **Derive device keys.** For every device, compute the EDK using hkds_server_generate_edk() with the newly created BDK and the device's DID. Store the EDK securely inside the terminal's secure element or microcontroller. Record the DID and KID in the server's enrolment database.
3. **Assign the initial KSN.** Construct a 16-byte KSN containing the device's DID and a 4-byte transaction counter initialized to zero. Persist the counter in non-volatile memory so that it increments across power cycles.
4. **Store the master keys securely.** The server should keep BDK and STK in a Hardware Security Module (HSM) or other tamper-resistant storage. Only the token encryption routines should have access to the STK; there is never a need to distribute STK outside the server.
5. **Plan for rotation.** Periodically rotate the STK (e.g., annually or after a fixed number of tokens). Because the EDK remains constant, rotating the STK does not require re-provisioning devices. When rolling the BDK (e.g., every few years), generate new EDKs for all devices; maintain overlap periods so that both old and new keys are accepted.

5 Integration into Payment Networks

5.1 Architecture

Payment networks require high-speed and robust key management for millions of terminals. HKDS eliminates the scaling bottlenecks of DUKPT by deriving transaction keys on demand instead of looking them up in a database. A typical deployment comprises:

- **Terminals (clients).** Each Point-of-Sale (POS) device contains its EDK and maintains a KSN counter. It generates token requests when its TKC runs out.
- **Authorization server (server).** The transaction processor holds the master key set (BDK, STK, KID) and a database mapping DIDs to EDKs. It responds to token requests and reconstructs transaction keys for message decryption.
- **Transport.** HKDS packets are carried over existing payment message channels (e.g., ISO 8583/ISO 20022) or over a dedicated secure transport such as VPN or SATP. Only

16-byte message blocks and small headers are added to the payload, so bandwidth overhead is minimal.

5.2 Integration Steps

1. **Device provisioning.** Manufacture or enroll terminals by generating an EDK for each device using the server's BDK and the devices DID. Write the EDK and initial KSN to the device's secure storage. At the server, record the device's DID, KSN and associated KID in the database.
2. **Token request cycle.** When the terminal's TKC is empty, it constructs a packet_token_request containing its current KSN and transmits it to the server. The server initializes a hkds_server_state with the client's KSN and master key set (hkds_server_initialize_state()), calls hkds_server_encrypt_token() to produce an encrypted token and MAC, and sends the result back.
3. **Token decryption and cache generation.** The terminal verifies and decrypts the received token using hkds_client_decrypt_token(). Upon success, it calls hkds_client_generate_cache() to produce a new TKC and resets its message counter. If verification fails, the terminal should request a new token.
4. **Encrypting transactions.** For each transaction (e.g., ISO 8583 message), the terminal retrieves the next key from the TKC and calls hkds_client_encrypt_authenticate_message() if a MAC is required or hkds_client_encrypt_message() for confidentiality only. The resulting ciphertext and tag are appended to the payment message.
5. **Server decryption.** The authorization server uses hkds_server_decrypt_verify_message() to verify the MAC and decrypt each message. It reconstructs the transaction key from the TKC computed on the fly and processes the payload. After a key is used, both client and server move to the next key in the cache.
6. **Counter management.** Ensure that the KSN counter increments correctly and does not wrap. Define a maximum number of token requests per device (e.g., 2^{32}) and schedule device re-enrolment before the counter exhausts. Persist the counter across power cycles to prevent key reuse.

5.3 Performance and Operational Considerations

- **Throughput:** HKDS uses only SHAKE and KMAC operations. Benchmarks indicate 4×–8× throughput improvement over DUKPT-AES, depending on the key size. This reduces transaction latency and server hardware requirements.

- **Scalability:** Because the server derives keys rather than storing per-device state, it can support millions of devices with constant memory. Token requests can be processed in parallel using the x8 or x64 functions to issue tokens or decrypt messages in batches.
- **Compliance:** HKDS meets PCI DSS guidelines (e.g., unique keys per transaction, strong key derivation functions) and relies on NIST-standardized primitives. It is designed for ease of FIPS 140-3 module certification when implemented within an HSM.

6 Integration into Cloud Platforms

6.1 Use Cases

While HKDS was designed for payment devices, its lightweight, hierarchical key derivation also lends itself to cloud environments where frequent key rotation is desirable. Potential applications include:

- **Secrets delivery:** Distribute short-lived secrets (API tokens, encryption keys) from a central secrets service to micro-services. Each micro-service holds an EDK; the secrets service acts as the HKDS server, issuing tokens that micro-services decrypt and use to derive working keys.
- **Service-to-service RPC:** Instead of long-lived TLS sessions, micro-services can wrap remote procedure calls in HKDS message packets. Because each call uses a unique key from the TKC and includes an optional MAC, replay and injection attacks are mitigated.
- **Tenant isolation:** In multi-tenant architectures, assign separate KIDs and STKs per tenant. Micro-services for each tenant generate their own EDKs derived from the corresponding BDK, preventing cross-tenant key reuse.

6.2 Integration Steps

1. **Key distribution service:** Integrate hkds_server_generate_mdk() into your secrets management pipeline. When a new micro-service registers, generate a master key set for its tenant and derive the micro-service's EDK using hkds_server_generate_edk(). Store the EDK in the service's configuration vault and the master keys in an HSM.
2. **Bootstrap token retrieval:** At start-up, the micro-service uses an authenticated channel (e.g., mTLS or instance metadata service) to request its first HKDS token from the secrets service. The secrets service initializes a server state with the micro-service's KSN and returns the encrypted token.
3. **Cache and encryption library:** Embed the HKDS client library into a middleware layer or sidecar. Upon receiving the token, call hkds_client_decrypt_token() and

hkds_client_generate_cache() to create a TKC. Intercept outbound messages: call hkds_client_encrypt_authenticate_message() to wrap payloads, including optional metadata (e.g., request ID) in the MAC.

4. **Stateless server functions:** For inbound requests, the secrets service (or an RPC gateway) uses the HKDS server library to reconstruct transaction keys on the fly. Because the server state does not retain per-message secrets, horizontal scaling is simplified; each instance can derive the same keys from the master key and KSN.
5. **Key rotation:** Rotate the STK on a defined schedule or after a breach. Because EDKs remain constant, micro-services simply request new tokens to refresh their caches. Use the KSN counter to manage token lifetime and prevent reuse.

6.3 Operational Considerations

- **Latency:** HKDS adds minimal overhead to RPC calls. Because key derivation occurs locally, there is no need to perform a full handshake once a token is obtained. Micro-services must only refresh their TKC periodically.
- **Auditing:** Log KSN values and token request events to trace which transaction keys were used. This provides accountability when investigating incidents.
- **Multi-region deployments:** Mirror the master key sets across regions in secure HSMs. Use region-specific KIDs to ensure tokens issued in one region are not accepted in another unless explicitly allowed.

7 Integration into SCADA Systems and Industrial Control

7.1 Requirements

Supervisory control and data acquisition (SCADA) and industrial control systems (ICS) demand real-time performance and resilience in harsh environments. HKDS's small code footprint and deterministic key derivation make it attractive for these applications.

7.2 Integration Steps

1. **Device EDK provisioning:** During manufacture, derive EDKs for programmable logic controllers (PLCs), remote terminal units (RTUs) and sensors using the plant's BDK and each devices DID. Store the EDK in non-volatile memory and record the DID in the control server.
2. **Token distribution:** Many industrial devices operate in networks with intermittent connectivity. Use HKDS to pre-compute multiple tokens for offline periods. The control

server can issue a batch of encrypted tokens and deliver them via secure channel or physically (e.g., USB or SD card) to the field device. The device decrypts each token as needed and fills its TKC.

3. **Secure command channel:** Wrap control commands (e.g., Modbus/TCP, DNP3) in HKDS message packets. Because each command uses a unique key and MAC, adversaries cannot replay or forge commands. The server uses hkds_server_decrypt_verify_message() to verify commands before acting upon them.
4. **Event reporting:** For telemetry data sent from devices to the SCADA head end, call hkds_client_encrypt_message() or hkds_client_encrypt_authenticate_message() to protect confidentiality and integrity. Use the optional MAC data field to bind context such as sensor ID and timestamp.
5. **Key rotation and maintenance:** When devices undergo scheduled maintenance, rotate the STK at the control server and generate fresh tokens for offline storage. Because devices keep their EDKs, they do not require hardware replacement.

7.3 Operational Considerations

- **Resource constraints:** HKDS's encryption and MAC operations are based on XOR and Keccak permutations. Even resource-constrained microcontrollers can implement them efficiently. The default message and tag sizes (16 bytes) fit into small buffers.
- **Deterministic timing:** All HKDS operations run in constant time, reducing susceptibility to timing attacks and making behavior predictable for real-time systems.
- **Physical security:** Because industrial devices are often in exposed locations, storing the EDK in a secure element (TPM, TRNG-backed microcontroller) is critical. The STK never leaves the control server.

8 Integration into Embedded and IoT Devices

8.1 Use Cases

IoT ecosystems involve vast numbers of constrained devices transmitting short messages (sensor readings, telemetry, control commands). HKDS fits these scenarios due to its small message size and elimination of heavy public-key operations during normal operation. It can be deployed for:

- **Smart meters and energy devices** where each reading must be authenticated and optionally encrypted.

- **Medical devices** that transmit patient data to a gateway; HKDS ensures confidentiality and integrity with minimal battery drain.
- **Vehicle networks** (e.g., CAN-bus overlays) that require deterministic timing and secure key rotation without re-provisioning modules.

8.2 Integration Steps

1. **EDK provisioning:** Use the same process as in SCADA: derive a per-device EDK from the BDK and the device's unique identifier. Write the EDK and an initial KSN into secure flash.
2. **Lightweight token exchange:** Because IoT devices may have intermittent connectivity, schedule periodic token requests when connectivity is available. Use the smallest protocol variant protocol_shake_128 for low-power devices, or protocol_shake_256 for mainstream security.
3. **Message protection:** Wrap outgoing telemetry or control messages in HKDS encrypted packets. Where payloads exceed 16 bytes, segment the data into multiple packets and include sequence numbers in the MAC data. Alternatively, combine HKDS with a tunnelling protocol (SATP) for streaming data.
4. **Gateway decryption:** Deploy the HKDS server library on a gateway or aggregator. Upon receiving a packet, the gateway reconstructs the transaction key from the EDK and STK, verifies the MAC and forwards the plaintext to the cloud or local network.
5. **Over-the-air updates:** Use HKDS to protect firmware updates delivered over the air. The server encrypts update fragments using the device's transaction keys; the device decrypts and authenticates each fragment before applying.

8.3 Operational Considerations

- **Battery life:** HKDS avoids expensive modular arithmetic. The primary cost is the Keccak permutation, which is efficient on 32-bit and 64-bit microcontrollers. Segmenting data into 16-byte blocks reduces memory requirements.
- **Network overhead:** Each HKDS packet adds only 4 bytes of header and 16 bytes of tag. This overhead is acceptable even on low-bit-rate links (LPWAN, Bluetooth LE).
- **Bootstrapping:** When devices are first deployed, ensure they can reach the HKDS server at least once to obtain a token. For fully offline devices, pre-load a sufficient number of tokens in manufacturing.

9 Security Best Practices and Operational Guidance

The strength of HKDS lies in its careful use of symmetric primitives and secure key management. To maximize security:

1. **Use a hardware RNG.** Provide a trusted random generator to hkds_server_generate_mdk() so that BDK and STK are unpredictable. For devices, generate unique DIDs using hardware entropy.
2. **Protect master keys.** Store the BDK and STK in an HSM. Only token-encryption routines should have access to the STK. Rotate the STK regularly; rotate the BDK less frequently, coordinating re-enrolment of devices.
3. **Verify MACs before decryption.** Always call hkds_client_decrypt_token() and hkds_server_decrypt_verify_message() to authenticate tokens and messages before using the decrypted data. Reject any packet with an invalid tag to prevent key guessing or modification attacks.
4. **Update counters properly.** Increment the KSN counter after each token exchange. Do not reuse a KSN; doing so would cause key reuse and break forward secrecy. Persist the counter to non-volatile storage.
5. **Use vectorized functions for high-throughput environments.** The HKDS library provides x8 and x64 versions of many functions that process messages or tokens in parallel. Use these functions when issuing tokens to many devices or processing large message batches to leverage SIMD acceleration.
6. **Integrate with hardware security modules.** Many payment processors and industrial controllers already use HSMs. Port the HKDS server routines into your HSM firmware and expose them via a secure API so that master keys never leave the secure boundary.
7. **Plan for certification.** Because HKDS relies on SHAKE and KMAC, implementations should target FIPS 140-3 compliance. Use the HKDS test suite to perform Known-Answer Tests, authenticated encryption tests and stress tests before deployment.

10 Conclusion

HKDS provides a robust and scalable solution to the long-standing challenges of symmetric key distribution. By harnessing post-quantum primitives and a hierarchical design, it enables millions of devices to derive unique transaction keys without server-side state. Practical integration is straightforward: generate master keys securely, derive device-specific EDKs, implement token request/response handling, and wrap messages with HKDS encryption and authentication functions. When combined with best practices for key storage, rotation and auditing, HKDS

delivers forward secrecy and quantum-safe assurance across payment networks, cloud services, industrial control and IoT deployments.