Cryptographic interfaces for secure IoT devices

September, 14, 2022

Kris Kwiatkowski
Sr. Cryptography Engineer, PQShield LTD
kris@pqshield.com

Plan

- Motivation: Usage of cryptographic APIs in the IoT
- An overview of the concepts used in handle-based APIs
- Lessons learned from integration of PQ schemes

Motivating Example

Root of Trust

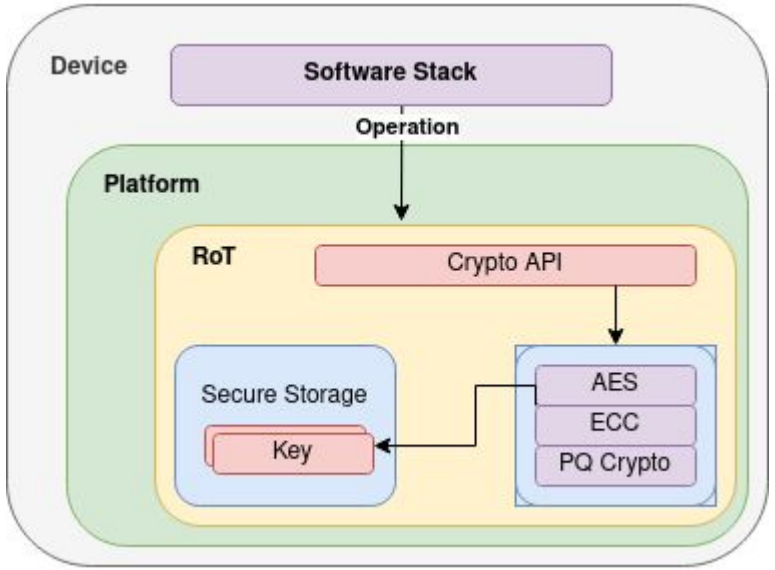
- **Need for hardware security**
  - Secure boot
  - Secure firmware update
  - Authentication to the cloud services
    - Requirement: be able to verify externally claims like device ID or SW/HW version in order to trust the device and provide access to resources
- **Root of Trust**
  - Hardware mechanism to protect sensitive data, away from the system software
  - Provides security services, secure storage, device binding…
  - Keeps private keys confidential and allows using those keys via trusted processing (symmetric & asymmetric cryptographic security services)
  - May contain cryptographic accelerator

Motivating Example

Root of Trust

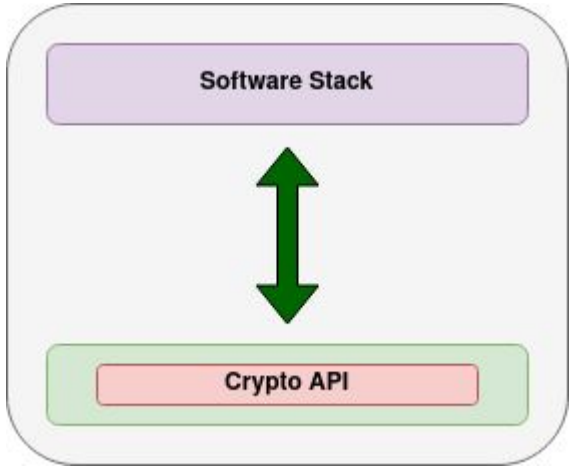
- **Usage of the RoT**
  - Secret material not accessible from outside the chip
  - Security services exposed via software interfaces

- **Properties of software interfaces**
  - Provide security services for trusted processing
  - Not possible to reveal the value of the secret kept by RoT
  - Ideally, a zero-overhead - device integrates only those interfaces that it requires



Application Programming Interface

- **What do we want from Cryptographic API**
  - Easy to learn and use by non-security experts
  - Flexible, allowing implementation of secure communication protocols
  - Preserving security properties
  - Hard to misuse
  - Well documented, with multiple examples



- **Diversity of IoT devices**
  - Different approach is required

Existing cryptographic APIs

- **OpenSSL**
  - Too large for small IoT devices
  - Focused on providing functionality for TLS on high-end CPUs
- **NaCL / libsodium**
  - Implementation of cryptographic primitives
  - Low level interfaces, don’t address cryptographic agility, not as flexible as needed
- **PKCS#11**
  - Complicated, not easy to use API
  - Quite big, IoT requires lighter API, a subset of PKCS#11
- **JCA / BouncyCastle / Rust / Go**
  - C is still the most used general-purpose language for embedded systems

Cryptographic APIs suitable for IoT devices

- PSA Certified

  PSA provides a recipe, based on industry best practice, that allows security to be consistently designed in, at both a hardware and firmware level.

  https://armmbed.github.io/mbed-crypto/html/

- Global Platform

  Designed, but with a focus on Trusted Execution Environment

  https://globalplatform.org/specs-library/tee-internal-core-api-specification

Design choices

- **A “keystore” interface:** define API of cryptographic operations, so that they work on a reference/handle to a key rather than a key material.
- **Unified API:** generically named functions allow performing cryptographic operations on a variety of algorithms of the same type.
- **Cryptographic operations separated from key management:** decoding of keys doesn’t interfere with actual cryptographic operation. Flexibility on key formats.
- **Possibility to implement subset of the API:** *”You only pay for what you use”*, makes the API suitable for both constrained devices and large backend systems.

PSA Crypto API

Use-case: ECDSA signature of pre-hashed message

Keystore-like interface

Secret key, accessible via handle and protected by hardware mechanisms

```
// Set attributes of an asymmetric key-pair based on NIST/p256
psa_key_id_t key_id;
psa_key_attributes_t attr = psa_key_attributes_init();
psa_set_key_type(&attr,
        PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&attr, 256);
psa_set_key_usage_flags(&attr,
        PSA_KEY_USAGE_SIGN_MESSAGE | PSA_KEY_USAGE_VERIFY_MESSAGE);
psa_set_key_algorithm(&attr, PSA_ALG_ECDSA_ANY);

// Generate keypair and assign it to the handle “key_id”
psa_generate_key(&attr, &key_id);
```

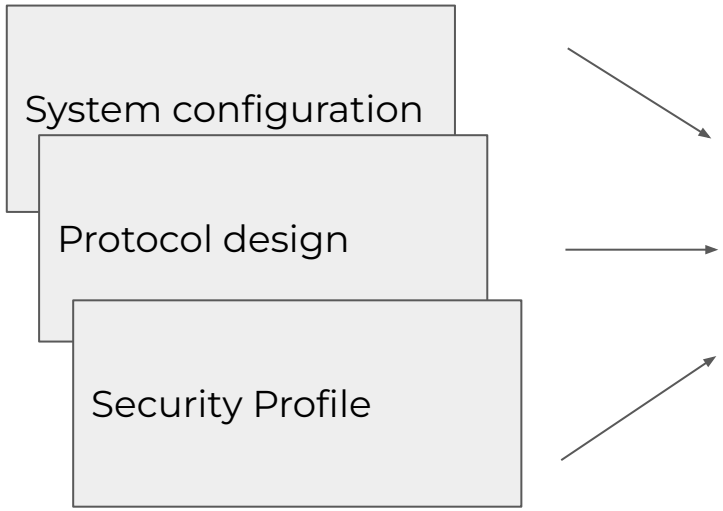
```
// Message signing
psa_sign_message(&key_id, PSA_ALG_ECDSA_ANY,
         msg_hash, sizeof(msg_hash),
         signature_buf, sizeof(signature_buf), &signature_len);
```

PSA Crypto API

Use-case: ECDSA signature of pre-hashed message

Unified API

Key attributes and algorithm IDs can be provided by the system configuration, negotiated by online protocol or loaded from the device's security profile.




```c
// Set attributes of an asymmetric key-pair based on NIST/p256
psa_key_id_t key_id;
psa_key_attributes_t attr = psa_key_attributes_init();
psa_set_key_type(&attr,
        PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&attr, 256);
psa_set_key_usage_flags(&attr,
        PSA_KEY_USAGE_SIGN_MESSAGE | PSA_KEY_USAGE_VERIFY_MESSAGE);
psa_set_key_algorithm(&attr, PSA_ALG_ECDSA_ANY);

// Generate keypair and assign it to the handle "key_id"
psa_generate_key(&attr, &key_id);
```

```c
// Message signing
psa_sign_message(&key_id, PSA_ALG_ECDSA_ANY,
         msg_hash, sizeof(msg_hash),
         signature_buf, sizeof(signature_buf), &signature_len);
```

PSA Crypto API

Use-case: ECDSA signature of pre-hashed message

Crypto operations separated from key management

No expensive operations on critical path

Flexibility on key formats and locations

```
// Set attributes of an asymmetric key-pair based on NIST/p256
psa_key_id_t key_id;
psa_key_attributes_t attr = psa_key_attributes_init();
psa_set_key_type(&attr,
        PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&attr, 256);
psa_set_key_usage_flags(&attr,
        PSA_KEY_USAGE_SIGN_MESSAGE | PSA_KEY_USAGE_VERIFY_MESSAGE);
psa_set_key_algorithm(&attr, PSA_ALG_ECDSA_ANY);

// Generate keypair and assign it to the handle “key_id”
psa_generate_key(&attr, &key_id);
```

```
// Message signing
psa_sign_message(&key_id, PSA_ALG_ECDSA_ANY,
         msg_hash, sizeof(msg_hash),
         signature_buf, sizeof(signature_buf), &signature_len);
```

```
// Message verification - load public key and verify signature
public_key[65] = {0x04, 0x8d, …};
psa_import_key(&attr, public_key, sizeof(public_key), &key_id);
psa_verify_hash(key_id, PSA_ALG_ECDSA_ANY,
     msg_hash, sizeof(msg_hash), signature_buf, signature_len);
```

Reusing concepts

- **Implementation details separated from API:** handle based APIs allows hiding cryptographic implementation from higher level code.
- **Unified API:** having generic structures and initialization of an algorithm by its ID promotes cryptographic agility
  - **Downside:** initialization of an algorithm requires heap allocation or uses pre-defined space on a stack
  - Implement only subset of algorithms to minimize stack usage and limit for number of keys used by the system.

```
    /* Dynamically allocated key data buffer.
     * Format as specified in psa_export_key(). */
    struct key_data
    {
        uint8_t *data;
        size_t bytes;
    } key;
} psa_key_slot_t;
```

Implementation of **psa_import_key()** in the mbedTLS

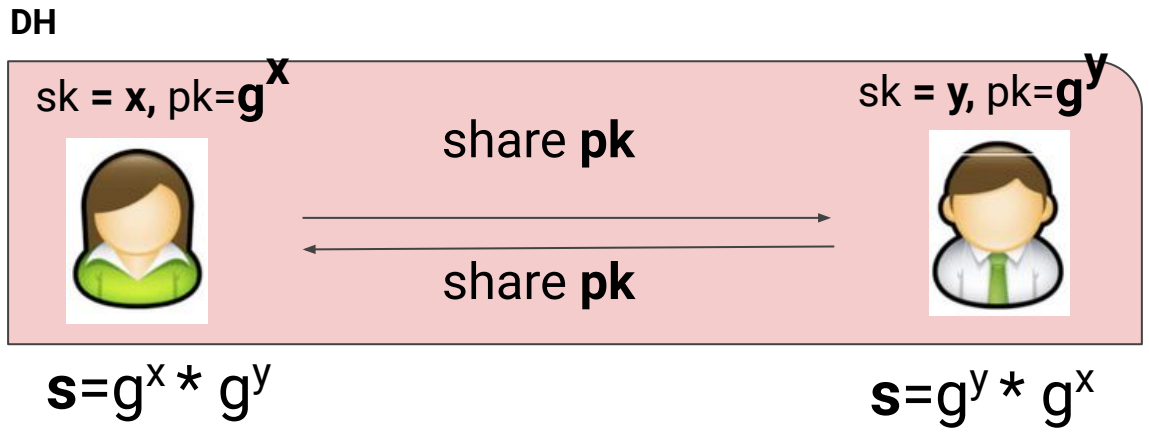
Changes required by PQ schemes

- Earlier this year, NIST selected for standardization 4 schemes resistant to **quantum computer** (CRCQ) attacks
- Those are:
  - Schemes for **digital signatures**
  - Scheme for Key Encapsulation Mechanism (**KEM**)
- Possible to reuse existing APIs for signature schemes
- **KEM** needs a different interface than the one most cryptographic libraries provide

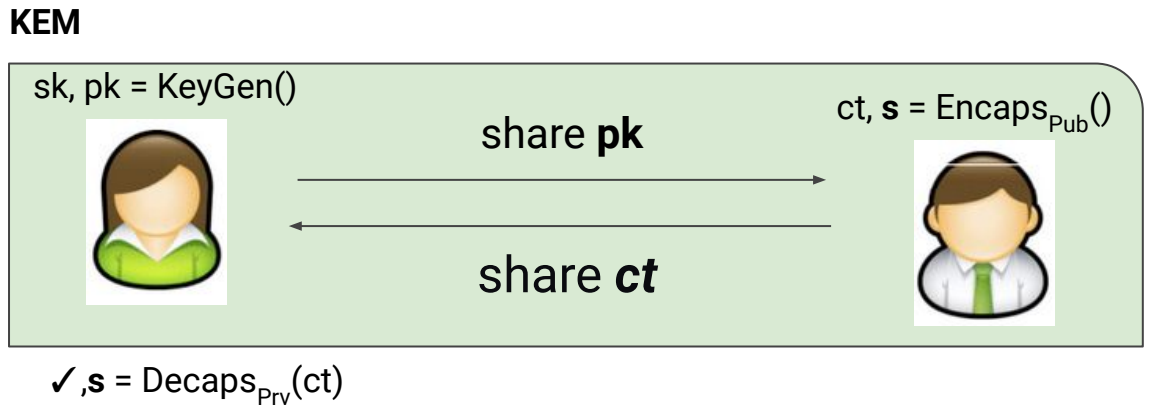
Key Encapsulation Mechanism

Diffie-Hellman key agreement vs KEM

- **KEM Interface**
  - Triple of algorithms: key generation, encapsulation, decapsulation
  - Asymmetric: Encapsulation outputs 2 results, Decapsulation output 1 result
  - Doesn't fit into DH interfaces



- **IND-CCA2 security**
  - Shared secret **s** always indistinguishable from random (even if attacker has an ability to decapsulate arbitrary ciphertexts).
  - Security against active attacker



KEM-interface for PSA Crypto

Post-quantum keypair

```
psa_key_id_t kem_key_id;
attr = psa_key_attributes_init();
psa_set_key_type(&attr, PSA_KEY_TYPE_KEM_KEY_PAIR);
psa_set_key_usage_flags(&attr, PSA_KEY_USAGE_ENCAPSULATE | PSA_KEY_USAGE_DECAPSULATE);
psa_set_key_algorithm(&attr, PSA_ALG_KYBER_512);
psa_generate_key(&attr, &kem_key_id);
```

Symmetric key attributes

```
secret_attr = psa_key_attributes_init();
psa_set_key_type(&secret_attr, PSA_KEY_TYPE_AES);
psa_set_key_usage_flags(&secret_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
psa_set_key_algorithm(&secret_attr, PSA_ALG_GCM);
psa_set_key_bits(&secret_attr, 256);
```

Encapsulation

```
psa_encapsulate(kem_key_id,
    &ciphertext_buf, cipherext_buf_len, &len, &secret_id);
```

Decapsulation

```
psa_decapsulate(kem_key_id,
    ciphertext_buf, cipherext_buf_len, &secret_id);
```

Encryption / Decryption

```
psa_aead_encrypt(&secret_id, PSA_ALG_GCM, …)
```

Post-quantum digital signatures

- PQ signature schemes sign the whole message rather than a hash (currently)

- No new functions required for non-incremental API
  - Adding key and algorithm IDs was enough
  - PSA provides a function psa_sign_message

- Nevertheless, incremental API is very useful.
  - Triple of functions is needed: `init()`, `update()`, `sign()/verify()`
  - Falcon: hash of the message is **prefixed** with a random nonce
  - Dilithium: prefix is hash of a public key (deterministic version of Dilithium)

Conclusion

- Popular libraries like OpenSSL not scalable for IoT devices
- PSA Crypto provides modular interfaces suitable for constrained devices
- PSA Cryptographic API
  - https://armmbed.github.io/mbed-crypto/html/
- GlobalPlatform TEE Core API
  - https://globalplatform.org/specs-library/tee-internal-core-api-specification/