Cryptnox SDK for C++ Manual

Release 1.0.0

June 20, 2026

Chapter 1

cryptnox-sdk-cpp

1.0.0.1 cryptnox-sdk-cpp

Platform-independent C++ core SDK for Cryptnox Hardware Wallet

`cryptnox-sdk-cpp` is the **shared C++ core SDK** for the **Cryptnox Hardware Wallet**. It implements the card-side protocol — secure channel establishment (SELECT → certificate → ECDH → mutual auth), APDU framing, PIN verification, signing, and user-data writing — independently of any target platform, NFC reader, or crypto library.

Important

> **This SDK is not usable on its own.** It exposes three abstract interfaces (`CW_NfcTransport`, `CW_CryptoProvider`, `CW_Logger`) that **must be implemented by a host integration**. It ships no transport driver, no crypto backend, and no logging output.

1.0.1 Used by

This core is consumed as a submodule by the platform-specific SDKs:

Integration	Repository
ESP32-S3 (ESP-IDF v5.5)	`cryptnox/cryptnox-sdk-esp32`
Arduino R4 (Renesas RA4M1)	`cryptnox/cryptnox-sdk-arduino`

If you want to talk to a Cryptnox card on real hardware, **start from one of those repositories**.

1.0.2 Porting to a new platform

This repository is intended as the **starting point for porting the SDK to a new platform** (another MCU family, a desktop OS, a different NFC reader, a different crypto backend, etc.).
A port consists of providing concrete implementations of the three adapter interfaces:

Interface	What you must provide
CW_NfcTransport	Driver for your NFC reader (PN532, PN7150, PC/SC, ...)
CW_CryptoProvider	SHA-256/512, AES-CBC, ECDH, EC key generation, RNG (mbedTLS, BearSSL, OpenSSL, hardware peripheral, ...)
CW_Logger	Output sink (UART, stdout, syslog, network, ...)

Then drop this repository in as a submodule (or copy of its sources) inside your project, build it together with your adapters, and instantiate CryptnoxWallet with the three injected dependencies. The existing platform SDKs (cryptnox-sdk-esp32, cryptnox-sdk-arduino) are useful references for a complete port.

1.0.3 What's inside

File	Role
CryptnoxWallet.{h,cpp}	High-level API: begin, connect, verifyPin, sign, writeUserData, disconnect
CW_SecureChannel.{h,cpp}	Secure channel protocol (mutual auth, session keys, encrypted APDU exchange)
CW_NfcTransport.h	**Adapter interface** — NFC reader contract
CW_CryptoProvider.h	**Adapter interface** — SHA-256/512, AES-CBC, ECDH, EC keygen, RNG
CW_Logger.h	**Adapter interface** — debug/serial output
CW_TrustedKeys.h	Cryptnox CA public keys used to verify card certificates
CW_Defs.h, CW_Utils.{h,cpp}	Constants, error codes, small helpers
platform_compat.h	Shim for non-Arduino targets

The three adapter interfaces are the only contract a host must satisfy. Everything else is self-contained.

1.0.4 Integrating the core

A host integration injects its three adapters into CryptnoxWallet:

```
#include "CryptnoxWallet.h"

// MyXxx = concrete adapters provided by the platform SDK:
//   - MyNfcTransport   : public CW_NfcTransport
//   - MyCryptoProvider : public CW_CryptoProvider
//   - MyLogger         : public CW_Logger

MyLogger          logger;
MyCryptoProvider  crypto;
MyNfcTransport    transport(/* platform-specific args */);
CryptnoxWallet    wallet(transport, logger, crypto);

if (!wallet.begin()) { /* reader init failed */ }

CW_SecureSession session;
if (wallet.connect(session)) {
    // wallet.verifyPin(...), wallet.sign(...), wallet.writeUserData(...)
    wallet.disconnect(session);
}
```

Full runnable examples (PIN verify, transaction signing, full ESP-IDF / Arduino boilerplate) live in the platform SDKs linked above.

1.0.5 Building standalone

There is **no standalone build target** in this repository. The CI workflow runs cppcheck static analysis only (.github/workflows/static_analysis.yml); host-supplied headers (uECC.h,

mbedtls/*, platform logger) are not vendored, so missingInclude is suppressed.
To compile and exercise the code, use one of the platform SDKs.

1.0.6 Documentation

The generated documentation for this project is available here.

1.0.7 License

cryptnox-sdk-cpp is dual-licensed:

- **LGPL-3.0** for open-source projects and proprietary projects that comply with LGPL requirements
- **Commercial license** for projects that require a proprietary license without LGPL obligations

For commercial inquiries, contact: contact@cryptnox.com

Chapter 2

Topic Documentation

2.1 Public API

Types and classes application code interacts with directly.

Classes

- struct CW_CardInfo

  Subset of the Cryptnox card info returned by APDU 0x80FA0000.
- struct CW_SignRequest

  Request parameters for CryptnoxWallet::sign.
- class CryptnoxWallet

  High-level interface for interacting with a Cryptnox Hardware Wallet over NFC.
- struct CW_SecureSession

  Holds cryptographic session state for reentrant secure channel operations.

2.1.1 Detailed Description

Types and classes application code interacts with directly.
Includes CryptnoxWallet (the main entry point), the sign request / result structs, CW_CardInfo, and CW_SecureSession.

2.2 Secure channel protocol

Low-level secure messaging implementation.

Classes

- class CW_SecureChannel

  Implements the Cryptnox secure channel protocol over NFC.

2.2.1 Detailed Description

Low-level secure messaging implementation.
CW_SecureChannel is composed inside CryptnoxWallet and is not normally used directly. Documented here for callers that need to drive the activation sequence manually (e.g. custom retry policies, fuzzing).

2.3 Adapter interfaces

Abstract contracts a host integration must implement.

Classes

- class CW_CryptoProvider

  Abstract interface for cryptographic operations used by CW_SecureChannel.
- class CW_Logger

  Abstract interface for serial/debug output.
- class CW_NfcTransport

  Abstract interface for NFC transport operations.
- class CW_Platform

  Abstract interface for platform-specific operations used by the SDK.

2.3.1 Detailed Description

Abstract contracts a host integration must implement.
Provide concrete implementations of CW_NfcTransport, CW_CryptoProvider, CW_Logger, and CW_Platform when porting the SDK to a new MCU or operating system.

2.4 Utilities & shared definitions

Platform-independent helpers and shared constants.

Classes

- class CW_Utils

  Portable utility functions for cryptographic and security operations.

Enumerations

- enum CW_Curve { CW_CURVE_SECP256R1 = 0 , CW_CURVE_SECP256K1 = 1 }

  Portable curve identifier used throughout the SDK.

2.4.1 Detailed Description

Platform-independent helpers and shared constants.
Includes CW_Utils (constant-time compare, secure wipe, safe memcpy, RNG), the `CW_CERT_*` / `CW_SIGN_*` error codes, and the trusted-CA key table (CW_TrustedKeys.h).

2.4.2 Enumeration Type Documentation

2.4.2.1 CW_Curve

`enum CW_Curve`
Portable curve identifier used throughout the SDK.
Replaces direct references to uECC_Curve_t at every API boundary so the abstract interfaces (CW_CryptoProvider, CW_SecureChannel) remain decoupled from any specific ECC back-end.

Enumerator

CW_CURVE_SECP256R1	NIST P-256 / secp256r1
CW_CURVE_SECP256K1	Koblitz secp256k1

Definition at line 151 of file CW_Defs.h.

Chapter 3

Class Documentation

3.1 __FlashStringHelper Class Reference

`#include <platform_compat.h>`

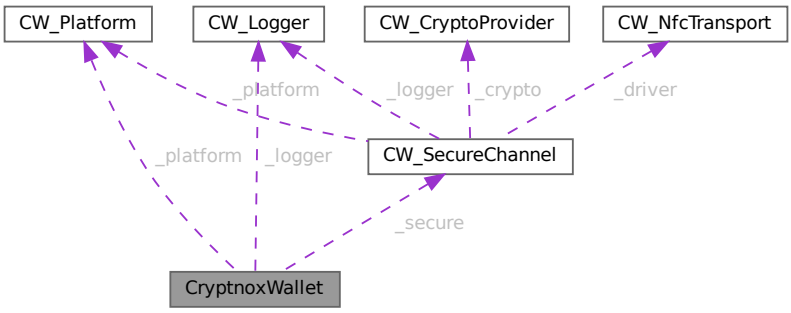
3.1.1 Detailed Description

Definition at line 41 of file platform_compat.h.
The documentation for this class was generated from the following file:

- platform_compat.h

3.2 CryptnoxWallet Class Reference

High-level interface for interacting with a Cryptnox Hardware Wallet over NFC.
`#include <CryptnoxWallet.h>`
Collaboration diagram for CryptnoxWallet:



Public Member Functions

- CryptnoxWallet (CW_NfcTransport &driver, CW_Logger &logger, CW_CryptoProvider &crypto, CW_Platform &platform)

  Construct a CryptnoxWallet.
- CryptnoxWallet (const CryptnoxWallet &)=delete
- CryptnoxWallet & operator= (const CryptnoxWallet &)=delete
- bool begin ()

Initialize the NFC module via the underlying transport driver.

- bool connect (CW_SecureSession &session)

  Connect to the Cryptnox card and establish a secure channel.
- bool establishSecureChannel (CW_SecureSession &session)

  Establish a secure channel (SELECT → certificate → ECDH → mutual auth).
- void disconnect (CW_SecureSession &session)

  Disconnect and securely clear the session.
- bool getCardInfo (CW_SecureSession &session, CW_CardInfo *info=NULL)

  Send a secured GET CARD INFO APDU (0x80FA0000) and optionally decode the owner name/email from the response.
- bool verifyPin (CW_SecureSession &session, const uint8_t *pin, uint8_t pinLength)

  Verify the PIN code on the card.
- CW_SignResult sign (CW_SignRequest &request)

  Sign a 32-byte digest using a card-resident key.
- bool writeUserData (CW_SecureSession &session, uint8_t slot, const uint8_t *data, uint16_t dataLength)

  Write data to a user memory slot, paginating in CW_USER_DATA_PAGE_SIZE chunks.

Static Public Member Functions

- static bool parseDerSignature (const uint8_t *der, uint8_t derLength, uint8_t *r, uint8_t &rLength, uint8_t *s, uint8_t &sLength)

  Parse a DER-encoded ECDSA signature to extract raw r and s values.

Private Member Functions

- bool isSecureChannelOpen (const CW_SecureSession &session) const
- bool printPN532FirmwareVersion ()
- bool validateSignRequest (const CW_SignRequest &request, CW_SignResult &result)
- void buildSignPayload (const CW_SignRequest &request, uint8_t *data, uint16_t &dataLength)
- bool sendSignApdu (CW_SignRequest &request, const uint8_t *data, uint16_t dataLength, uint8_t *derResponse, uint16_t &derLength, CW_SignResult &result)
- bool extractRawSignature (const uint8_t *derResponse, uint16_t derLength, CW_SignResult &result)
- void debugPrintSignature (const uint8_t *signature)

Private Attributes

- CW_Logger & _logger

  Logging interface.
- CW_Platform & _platform

  Platform abstraction (sleep_ms).
- CW_SecureChannel _secure

  Owned secure channel.

3.2.1 Detailed Description

High-level interface for interacting with a Cryptnox Hardware Wallet over NFC.
Manages card connection, secure channel establishment (delegated to CW_SecureChannel), PIN verification, transaction signing, user-data writing, and card-info retrieval.
Dependencies are injected by the caller via the constructor. The class itself only talks to the four abstract adapters, keeping the implementation platform-independent.

Typical lifecycle

```
CryptnoxWallet wallet(transport, logger, crypto, platform);
wallet.begin();

CW_SecureSession session;
if (wallet.connect(session)) {
    wallet.verifyPin(session, pin, pinLen);
    wallet.sign(req);
}
wallet.disconnect(session);   // mandatory -- even on connect() failure
```

Note

Single-shot use — the class is non-copyable. Reuse the same CryptnoxWallet instance across multiple card sessions; do not construct one per APDU.

Definition at line 160 of file CryptnoxWallet.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 CryptnoxWallet() [1/2]

```
CryptnoxWallet::CryptnoxWallet (
            CW_NfcTransport & driver,
            CW_Logger & logger,
            CW_CryptoProvider & crypto,
            CW_Platform & platform)
```

Construct a CryptnoxWallet.

Parameters

driver	Reference to the NFC transport implementation.
logger	Reference to the logging implementation.
crypto	Reference to the crypto provider implementation.
platform	Reference to the platform abstraction (for sleep_ms).

Definition at line 27 of file CryptnoxWallet.cpp.
References _logger, _platform, and _secure.
Referenced by CryptnoxWallet(), and operator=().

3.2.2.2 CryptnoxWallet() [2/2]

```
CryptnoxWallet::CryptnoxWallet (
            const CryptnoxWallet & )  [delete]
```

References CryptnoxWallet().

3.2.3 Member Function Documentation

3.2.3.1 begin()

```
bool CryptnoxWallet::begin ()
```

Initialize the NFC module via the underlying transport driver.

Returns

true if the module was successfully initialised, false otherwise.

Definition at line 36 of file CryptnoxWallet.cpp.
References _secure, and printPN532FirmwareVersion().

3.2.3.2 buildSignPayload()

```
void CryptnoxWallet::buildSignPayload (
            const CW_SignRequest & request,
            uint8_t * data,
            uint16_t & dataLength)  [private]
```

Definition at line 431 of file CryptnoxWallet.cpp.
References CW_HASH_SIZE, CW_MAX_DERIVE_PATH_LENGTH, CW_MAX_PIN_LENGTH, CW_SIGN_DERIVE_K1, CW_SIGN_DERIVE_R1, CW_SignRequest::derivePath, CW_SignRequest::derivePathLength CW_SignRequest::hash, CW_SignRequest::hashLength, CW_SignRequest::keyType, CW_SignRequest::pin, CW_SignRequest::pinLessMode, and CW_Utils::safe_memcpy().
Referenced by sign().

3.2.3.3 connect()

```
bool CryptnoxWallet::connect (
            CW_SecureSession & session)
```

Connect to the Cryptnox card and establish a secure channel.
Retries the full card activation sequence up to `CW_CONNECT_MAX_ATTEMPTS` times. On any failure (including transient transport errors) the session is securely wiped before the next attempt so no partial key material can survive a retry (CRIT-04).

Parameters

`out`	*session*	Secure session to populate with derived keys and IV on success; left zero-wiped on failure.

Returns

true if the secure channel was established and `session` is ready for use, false otherwise.

Postcondition

On true: `session` holds valid Kenc / Kmac / IV.

On false: `session` is zero-wiped.

Warning

Always call disconnect() after this — even on failure — to release the reader for the next card cycle.

Definition at line 44 of file CryptnoxWallet.cpp.
References _logger, _platform, _secure, CW_SecureSession::clear(), CW_CONNECT_MAX_ATTEMPTS, establishSecureChannel(), and F.

3.2.3.4 debugPrintSignature()

```
void CryptnoxWallet::debugPrintSignature (
            const uint8_t * signature)  [private]
```

Definition at line 543 of file CryptnoxWallet.cpp.
References _logger, CW_RAW_SIGNATURE_SIZE, F, and HEX.
Referenced by sign().

3.2.3.5 disconnect()

```
void CryptnoxWallet::disconnect (
            CW_SecureSession & session)
```

Disconnect and securely clear the session.
Wipes any session keys and resets the NFC reader so the next card detection cycle starts from a clean state.

Parameters

in,out	*session*	Session to clear. Safe to pass a never-connected or partially-connected session.

Precondition

Must be called at the end of every card-processing iteration — including iterations where connect() failed — otherwise the NFC reader may remain in an unresponsive state.

Definition at line 158 of file CryptnoxWallet.cpp.
References _secure, CW_SecureSession::clear(), and isSecureChannelOpen().

3.2.3.6 establishSecureChannel()

```
bool CryptnoxWallet::establishSecureChannel (
            CW_SecureSession & session)
```

Establish a secure channel (SELECT → certificate → ECDH → mutual auth).
Lower-level than connect(): runs the full activation sequence once, without the retry loop. Used internally by connect(); exposed for advanced callers that handle retry policy themselves.

Parameters

out	*session*	Secure session to populate.

Returns

true if mutual authentication succeeded, false if any step of the activation sequence (SELECT, certificate chain verification, ECDH, MAC check) failed.

Warning

All sensitive ephemeral key material is wiped from the stack on every exit path (H-01, M-02).

Definition at line 75 of file CryptnoxWallet.cpp.
References _logger, _secure, CW_CERT_OK, CW_CURVE_SECP256R1, F, HEX, and CW_Utils::secure_wipe().
Referenced by connect().

3.2.3.7 extractRawSignature()

```
bool CryptnoxWallet::extractRawSignature (
            const uint8_t * derResponse,
            uint16_t derLength,
            CW_SignResult & result)  [private]
```

Definition at line 480 of file CryptnoxWallet.cpp.
References _logger, CW_DER_TAG_SEQUENCE, CW_NOK, CW_RAW_SIGNATURE_SIZE, CW_SignResult::errorCode, F, parseDerSignature(), CW_Utils::safe_memcpy(), CW_Utils::secure_wipe(), and CW_SignResult::signature.
Referenced by sign().

3.2.3.8 getCardInfo()

```
bool CryptnoxWallet::getCardInfo (
            CW_SecureSession & session,
            CW_CardInfo * info = NULL)
```

Send a secured GET CARD INFO APDU (0x80FA0000) and optionally decode the owner name/email from the response.

Parameters

in, out	*session*	Valid secure session.
out	*info*	Optional output. When non-NULL and the call succeeds, populated with the card's owner name and email (ASCII, NUL-terminated).

Returns

true if the secure exchange completed and (when `info` is non-NULL) parsing the name/email fields succeeded.

Definition at line 165 of file CryptnoxWallet.cpp.
References _logger, _secure, CW_CARD_EMAIL_MAX_LEN, CW_CARD_NAME_MAX_LEN, CW_CardInfo::email, F, isSecureChannelOpen(), CW_CardInfo::name, CW_Utils::safe_memcpy(), and CW_Utils::secure_wipe().

3.2.3.9 isSecureChannelOpen()

```
bool CryptnoxWallet::isSecureChannelOpen (
            const CW_SecureSession & session) const  [private]
```

Definition at line 370 of file CryptnoxWallet.cpp.
References CW_SecureSession::aesKey, and CW_AESKEY_SIZE.
Referenced by disconnect(), getCardInfo(), validateSignRequest(), verifyPin(), and writeUserData().

3.2.3.10 operator=()

```
CryptnoxWallet & CryptnoxWallet::operator= (
            const CryptnoxWallet & )  [delete]
```

References CryptnoxWallet().

3.2.3.11 parseDerSignature()

```
bool CryptnoxWallet::parseDerSignature (
            const uint8_t * der,
            uint8_t derLength,
            uint8_t * r,
            uint8_t & rLength,
            uint8_t * s,
            uint8_t & sLength)  [static]
```

Parse a DER-encoded ECDSA signature to extract raw r and s values.

Parameters

in	*der*	DER-encoded signature bytes.
in	*derLength*	DER length.
out	*r*	Buffer for r (at least 33 bytes).
out	*rLength*	Actual r length written.
out	*s*	Buffer for s (at least 33 bytes).
out	*sLength*	Actual s length written.

Returns

true on success, false on malformed DER.

Definition at line 322 of file CryptnoxWallet.cpp.
References CW_DER_TAG_INTEGER, CW_DER_TAG_SEQUENCE, and CW_Utils::safe_memcpy().
Referenced by extractRawSignature().

3.2.3.12 printPN532FirmwareVersion()

```
bool CryptnoxWallet::printPN532FirmwareVersion ()  [private]
```

Definition at line 378 of file CryptnoxWallet.cpp.
References _secure.
Referenced by begin().

3.2.3.13 sendSignApdu()

```
bool CryptnoxWallet::sendSignApdu (
            CW_SignRequest & request,
            const uint8_t * data,
            uint16_t dataLength,
            uint8_t * derResponse,
            uint16_t & derLength,
            CW_SignResult & result)  [private]
```

Definition at line 456 of file CryptnoxWallet.cpp.
References _logger, _secure, CW_SIGN_NO_KEY_LOADED, CW_SignResult::errorCode, F, CW_SignRequest::keyType, CW_SignRequest::session, and CW_SignRequest::signatureType.
Referenced by sign().

3.2.3.14 sign()

```
CW_SignResult CryptnoxWallet::sign (
            CW_SignRequest & request)
```

Sign a 32-byte digest using a card-resident key.
Builds the SIGN payload (hash || optional BIP32 path || optional PIN), sends it through the secure channel, parses the DER signature returned by the card, and unpacks it into the canonical 64-byte raw form (r[32] || s[32]).

Parameters

in	*request*	Sign parameters — must reference a valid secure session, a 32-byte hash, the desired key/signature type, and the PIN (unless `pinLessMode` is set).

Returns

CW_SignResult. On success `errorCode` is CW_OK and `signature` holds the 64-byte raw signature. On failure the signature is zeroed and `errorCode` indicates the cause:

Return values

CW_OK	Signature valid.
CW_INVALID_SESSION	Secure channel not open.
CW_SIGN_KEY_TOO_SHORT	Bad hash buffer / length.
CW_SIGN_NO_KEY_LOADED	Card rejected the SIGN APDU.
CW_SIGN_PIN_INCORRECT	PIN length out of range.
CW_SIGN_KEY_TOO_SHORT_WITH_PINLESS_MODE	PIN-less mode requested but `keyType` is not CW_SIGN_PINLESS_K1.

Warning

When `pinLessMode` is false the `request.pin` field must be populated. The destructor of CW_SignRequest securely wipes the PIN, but the caller must zero any other copy.

Definition at line 293 of file CryptnoxWallet.cpp.
References buildSignPayload(), CW_HASH_SIZE, CW_MAX_DERIVE_PATH_LENGTH, CW_MAX_PIN_LENGTH, CW_OK, debugPrintSignature(), CW_SignResult::errorCode, extractRawSignature(), CW_Utils::secure_wipe(), sendSignApdu(), CW_SignResult::signature, and validateSignRequest().

3.2.3.15 validateSignRequest()

```
bool CryptnoxWallet::validateSignRequest (
            const CW_SignRequest & request,
            CW_SignResult & result)  [private]
```

Definition at line 382 of file CryptnoxWallet.cpp.
References _logger, CW_HASH_SIZE, CW_INVALID_SESSION, CW_MAX_PIN_LENGTH, CW_MIN_PIN_LENGTH, CW_SIGN_KEY_TOO_SHORT, CW_SIGN_KEY_TOO_SHORT_WITH_PINLESS_MODE, CW_SIGN_PIN_INCORRECT, CW_SIGN_PINLESS_K1, CW_SignResult::errorCode, F, CW_SignRequest::hash, CW_SignRequest::hashLength, isSecureChannelOpen(), CW_SignRequest::keyType, CW_SignRequest::pin, CW_SignRequest::pinLessMode, and CW_SignRequest::session.
Referenced by sign().

3.2.3.16 verifyPin()

```
bool CryptnoxWallet::verifyPin (
            CW_SecureSession & session,
            const uint8_t * pin,
            uint8_t pinLength)
```

Verify the PIN code on the card.
Sends an encrypted VERIFY PIN APDU. The card maintains a try counter: every wrong attempt decrements it, and reaching zero locks the PIN permanently until a successful PUK / re-initialisation flow.

Parameters

in,out	*session*	Valid secure session.
in	*pin*	PIN bytes (ASCII digits, 4–9 characters).
in	*pinLength*	Length of the PIN (must be in [CW_MIN_PIN_LENGTH, CW_MAX_PIN_LENGTH]).

Returns

true if the card accepted the PIN, false on wrong PIN, closed or invalid session, length out of range, or transport / MAC failure.

Warning

Each failed attempt decrements the card's PIN counter. Treat a false return as "wrong PIN" only after confirming session validity — a transport glitch should not be retried with a new PIN.

Definition at line 219 of file CryptnoxWallet.cpp.
References _logger, _secure, CW_MAX_PIN_LENGTH, CW_MIN_PIN_LENGTH, F, isSecureChannelOpen(), CW_Utils::safe_memcpy(), and CW_Utils::secure_wipe().

3.2.3.17 writeUserData()

```
bool CryptnoxWallet::writeUserData (
            CW_SecureSession & session,
            uint8_t slot,
            const uint8_t * data,
            uint16_t dataLength)
```

Write data to a user memory slot, paginating in CW_USER_DATA_PAGE_SIZE chunks.

Parameters

in,out	*session*	Valid secure session.
in	*slot*	User data slot index.
in	*data*	Data to write.
in	*dataLength*	Total bytes to write.

Returns

true if all pages written successfully, false otherwise.

Definition at line 241 of file CryptnoxWallet.cpp.
References _logger, _secure, CW_USER_DATA_PAGE_SIZE, F, and isSecureChannelOpen().

3.2.4 Member Data Documentation

3.2.4.1 _logger

```
CW_Logger& CryptnoxWallet::_logger  [private]
```

Logging interface.
Definition at line 328 of file CryptnoxWallet.h.
Referenced by connect(), CryptnoxWallet(), debugPrintSignature(), establishSecureChannel(), extractRawSignature(), getCardInfo(), sendSignApdu(), validateSignRequest(), verifyPin(), and writeUserData().

3.2.4.2 _platform

```
CW_Platform& CryptnoxWallet::_platform  [private]
```

Platform abstraction (sleep_ms).
Definition at line 329 of file CryptnoxWallet.h.
Referenced by connect(), and CryptnoxWallet().

3.2.4.3 _secure

```
CW_SecureChannel CryptnoxWallet::_secure  [private]
```

Owned secure channel.
Definition at line 330 of file CryptnoxWallet.h.
Referenced by begin(), connect(), CryptnoxWallet(), disconnect(), establishSecureChannel(), getCardInfo(), printPN532FirmwareVersion(), sendSignApdu(), verifyPin(), and writeUserData().
The documentation for this class was generated from the following files:

- CryptnoxWallet.h
- CryptnoxWallet.cpp

3.3 CW_CardInfo Struct Reference

Subset of the Cryptnox card info returned by APDU 0x80FA0000.
`#include <CryptnoxWallet.h>`

Public Member Functions

- CW_CardInfo ()

Public Attributes

- char name [CW_CARD_NAME_MAX_LEN+1U]
- char email [CW_CARD_EMAIL_MAX_LEN+1U]

3.3.1 Detailed Description

Subset of the Cryptnox card info returned by APDU 0x80FA0000.
Mirrors the fields the Python SDK exposes as `card._owner`: ASCII name and email programmed when the card was initialised. NUL-terminated.
Definition at line 55 of file CryptnoxWallet.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 CW_CardInfo()

```
CW_CardInfo::CW_CardInfo () [inline]
```
Definition at line 59 of file CryptnoxWallet.h.
References email, and name.

3.3.3 Member Data Documentation

3.3.3.1 email

```
char CW_CardInfo::email[CW_CARD_EMAIL_MAX_LEN+1U]
```
NUL-terminated ASCII email.
Definition at line 57 of file CryptnoxWallet.h.
Referenced by CW_CardInfo(), and CryptnoxWallet::getCardInfo().

3.3.3.2 name

```
char CW_CardInfo::name[CW_CARD_NAME_MAX_LEN+1U]
```
NUL-terminated ASCII name.
Definition at line 56 of file CryptnoxWallet.h.
Referenced by CW_CardInfo(), and CryptnoxWallet::getCardInfo().
The documentation for this struct was generated from the following file:

- CryptnoxWallet.h

3.4 CW_CryptoProvider Class Reference

Abstract interface for cryptographic operations used by CW_SecureChannel.
`#include <CW_CryptoProvider.h>`

Public Member Functions

- virtual bool sha256 (const uint8_t *data, size_t len, uint8_t *out)=0
  Compute SHA-256 over a contiguous data buffer.
- virtual bool sha512 (const uint8_t *data, size_t len, uint8_t *out)=0
  Compute SHA-512 over a contiguous data buffer.
- virtual uint16_t aesCbcEncrypt (const uint8_t *in, uint16_t len, uint8_t *out, const uint8_t *key, uint8_t keyLen, uint8_t *iv, bool bitPadding)=0
  AES-CBC encrypt.
- virtual uint16_t aesCbcDecrypt (uint8_t *in, uint16_t len, uint8_t *out, const uint8_t *key, uint8_t keyLen, uint8_t *iv, bool bitPadding)=0
  AES-CBC decrypt.
- virtual bool ecdh (const uint8_t *pubKey, const uint8_t *privKey, uint8_t *secret, CW_Curve curve)=0
  ECDH shared secret computation.
- virtual bool makeKey (uint8_t *pubKey, uint8_t *privKey, CW_Curve curve)=0
  Generate a new EC key pair.

- virtual bool random (uint8_t *dest, unsigned size)=0

  Fill a buffer with cryptographically random bytes.
- virtual bool ecdsaVerify (const uint8_t *pubKey64, const uint8_t *hash, size_t hashLen, const uint8_t *sig, CW_Curve curve)=0

  Verify an ECDSA signature (raw r||s, 64 bytes) against a message hash.
- virtual ~CW_CryptoProvider ()

3.4.1 Detailed Description

Abstract interface for cryptographic operations used by CW_SecureChannel.
Decouples the secure channel protocol from any specific crypto library. The concrete ESP32 implementation (ESP32CryptoProvider) wraps mbedTLS (SHA-256/SHA-512/AES-CBC hardware-accelerated on ESP32-S3) and the ESP32 hardware TRNG for random number generation.
Definition at line 45 of file CW_CryptoProvider.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 ~CW_CryptoProvider()

```
virtual CW_CryptoProvider::~CW_CryptoProvider ()  [inline], [virtual]
```

Definition at line 146 of file CW_CryptoProvider.h.

3.4.3 Member Function Documentation

3.4.3.1 aesCbcDecrypt()

```
virtual uint16_t CW_CryptoProvider::aesCbcDecrypt (
            uint8_t * in,
            uint16_t len,
            uint8_t * out,
            const uint8_t * key,
            uint8_t keyLen,
            uint8_t * iv,
            bool bitPadding)  [pure virtual]
```

AES-CBC decrypt.

Parameters

in	*in*	Ciphertext input buffer (non-const; may be modified internally).
in	*len*	Length of the ciphertext.
out	*out*	Plaintext output buffer.
in	*key*	AES key bytes.
in	*keyLen*	AES key length in bytes.
in,out	*iv*	16-byte IV used as decrypt IV.
in	*bitPadding*	true = Bit padding removal; false = Null padding (no removal).

Returns

Length of the plaintext written to out.

3.4.3.2 aesCbcEncrypt()

```
virtual uint16_t CW_CryptoProvider::aesCbcEncrypt (
            const uint8_t * in,
            uint16_t len,
            uint8_t * out,
```

```
            const uint8_t * key,
            uint8_t keyLen,
            uint8_t * iv,
            bool bitPadding)  [pure virtual]
```

AES-CBC encrypt.

Parameters

in	*in*	Plaintext input buffer.
in	*len*	Length of the plaintext.
out	*out*	Ciphertext output buffer (must be large enough for padding).
in	*key*	AES key bytes.
in	*keyLen*	AES key length in bytes (16, 24, or 32).
in,out	*iv*	16-byte IV; updated to last cipher block on return.
in	*bitPadding*	true = ISO/IEC 9797-1 Method 2 (Bit) padding; false = Null padding (no padding added).

Returns

Length of the ciphertext written to out.

3.4.3.3 ecdh()

```
virtual bool CW_CryptoProvider::ecdh (
            const uint8_t * pubKey,
            const uint8_t * privKey,
            uint8_t * secret,
            CW_Curve curve)  [pure virtual]
```

ECDH shared secret computation.

Parameters

in	*pubKey*	Remote public key (64 bytes, X||Y, no 0x04 prefix).
in	*privKey*	Local private key (32 bytes).
out	*secret*	32-byte shared secret output.
in	*curve*	Curve identifier (CW_CURVE_SECP256R1 or CW_CURVE_SECP256K1).

Returns

true on success, false otherwise.

3.4.3.4 ecdsaVerify()

```
virtual bool CW_CryptoProvider::ecdsaVerify (
            const uint8_t * pubKey64,
            const uint8_t * hash,
            size_t hashLen,
            const uint8_t * sig,
            CW_Curve curve)  [pure virtual]
```

Verify an ECDSA signature (raw r||s, 64 bytes) against a message hash.

Parameters

in	*pubKey64*	64-byte public key (X||Y, no 0x04 prefix).
in	*hash*	Message hash buffer.
in	*hashLen*	Length of the hash in bytes.
in	*sig*	64-byte raw signature (r[32]||s[32]).
in	*curve*	Curve identifier for the verification operation.

Returns

true if the signature is valid, false otherwise.

3.4.3.5 makeKey()

```
virtual bool CW_CryptoProvider::makeKey (
            uint8_t * pubKey,
            uint8_t * privKey,
            CW_Curve curve)  [pure virtual]
```

Generate a new EC key pair.

Parameters

out	*pubKey*	64-byte public key output (X||Y, no prefix).
out	*privKey*	32-byte private key output.
in	*curve*	Curve identifier (CW_CURVE_SECP256R1 or CW_CURVE_SECP256K1).

Returns

true on success, false otherwise.

3.4.3.6 random()

```
virtual bool CW_CryptoProvider::random (
            uint8_t * dest,
            unsigned size)  [pure virtual]
```

Fill a buffer with cryptographically random bytes.

Parameters

out	*dest*	Buffer to fill.
in	*size*	Number of bytes to generate.

Returns

true on success, false otherwise.

3.4.3.7 sha256()

```
virtual bool CW_CryptoProvider::sha256 (
            const uint8_t * data,
            size_t len,
            uint8_t * out)  [pure virtual]
```

Compute SHA-256 over a contiguous data buffer.

Parameters

in	*data*	Input buffer.
in	*len*	Number of bytes to hash.
out	*out*	32-byte output buffer.

Returns

true on success, false if the underlying hash accelerator faults.

3.4.3.8 sha512()

```
virtual bool CW_CryptoProvider::sha512 (
            const uint8_t * data,
            size_t len,
            uint8_t * out)  [pure virtual]
```

Compute SHA-512 over a contiguous data buffer.

Parameters

in	*data*	Input buffer.
in	*len*	Number of bytes to hash.
out	*out*	64-byte output buffer.

Returns

true on success, false if the underlying hash accelerator faults.

The documentation for this class was generated from the following file:

- CW_CryptoProvider.h

3.5 CW_Logger Class Reference

Abstract interface for serial/debug output.

`#include <CW_Logger.h>`

Public Member Functions

- virtual bool begin (unsigned long baudRate=115200UL)=0

  Initialize the logging interface.
- virtual ~CW_Logger ()

Print methods (no newline)

- virtual void print (const __FlashStringHelper *str)=0
- virtual void print (const char *str)=0
- virtual void print (char c)=0
- virtual void print (uint8_t value, int base=DEC)=0
- virtual void print (uint16_t value, int base=DEC)=0
- virtual void print (uint32_t value, int base=DEC)=0
- virtual void print (int value, int base=DEC)=0

Println methods (with newline)

- virtual void println ()=0

- virtual void println (const __FlashStringHelper *str)=0
- virtual void println (const char *str)=0
- virtual void println (char c)=0
- virtual void println (uint8_t value, int base=DEC)=0
- virtual void println (uint16_t value, int base=DEC)=0
- virtual void println (uint32_t value, int base=DEC)=0
- virtual void println (int value, int base=DEC)=0

3.5.1 Detailed Description

Abstract interface for serial/debug output.
Provides a hardware-agnostic logging contract so that higher-level components (CryptnoxWallet, CW_SecureChannel) remain independent of the physical output device (UART, LCD, network, etc.).
On Arduino, the F() macro returns a __FlashStringHelper* so the dedicated overloads are called, keeping string literals in flash. On non-Arduino, F() is the identity macro (returns const char*), so the print(const char*) overload is called instead.
Definition at line 48 of file CW_Logger.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 ~CW_Logger()

```
virtual CW_Logger::~CW_Logger ()  [inline], [virtual]
```

Definition at line 80 of file CW_Logger.h.

3.5.3 Member Function Documentation

3.5.3.1 begin()

```
virtual bool CW_Logger::begin (
            unsigned long baudRate = 115200UL)  [pure virtual]
```

Initialize the logging interface.

Parameters

baudRate	Baud rate (relevant for UART implementations).

Returns

true if initialization succeeded, false otherwise.

3.5.3.2 print() [1/7]

```
virtual void CW_Logger::print (
            char c)  [pure virtual]
```

3.5.3.3 print() [2/7]

```
virtual void CW_Logger::print (
            const __FlashStringHelper * str)  [pure virtual]
```

3.5.3.4 print() [3/7]

```
virtual void CW_Logger::print (
            const char * str)  [pure virtual]
```

3.5.3.5 print() [4/7]

```
virtual void CW_Logger::print (
            int value,
            int base = DEC)  [pure virtual]
```

References DEC.

3.5.3.6 print() [5/7]

```
virtual void CW_Logger::print (
            uint16_t value,
            int base = DEC)  [pure virtual]
```

References DEC.

3.5.3.7 print() [6/7]

```
virtual void CW_Logger::print (
            uint32_t value,
            int base = DEC)  [pure virtual]
```

References DEC.

3.5.3.8 print() [7/7]

```
virtual void CW_Logger::print (
            uint8_t value,
            int base = DEC)  [pure virtual]
```

References DEC.

3.5.3.9 println() [1/8]

```
virtual void CW_Logger::println ()  [pure virtual]
```

3.5.3.10 println() [2/8]

```
virtual void CW_Logger::println (
            char c)  [pure virtual]
```

3.5.3.11 println() [3/8]

```
virtual void CW_Logger::println (
            const __FlashStringHelper * str)  [pure virtual]
```

3.5.3.12 println() [4/8]

```
virtual void CW_Logger::println (
            const char * str)  [pure virtual]
```

3.5.3.13 println() [5/8]

```
virtual void CW_Logger::println (
            int value,
            int base = DEC)  [pure virtual]
```

References DEC.

3.5.3.14 println() [6/8]

```
virtual void CW_Logger::println (
            uint16_t value,
            int base = DEC)  [pure virtual]
```

References DEC.

3.5.3.15 println() [7/8]

```
virtual void CW_Logger::println (
            uint32_t value,
            int base = DEC)  [pure virtual]
```

References DEC.

3.5.3.16 println() [8/8]

```
virtual void CW_Logger::println (
            uint8_t value,
            int base = DEC)  [pure virtual]
```

References DEC.
The documentation for this class was generated from the following file:

- CW_Logger.h

3.6 CW_NfcTransport Class Reference

Abstract interface for NFC transport operations.
`#include <CW_NfcTransport.h>`

Public Member Functions

- virtual bool begin ()=0

  Initialize the NFC transport hardware.
- virtual bool inListPassiveTarget ()=0

  Detect the presence of a passive ISO-DEP NFC target.
- virtual bool sendAPDU (const uint8_t *apdu, uint8_t apduLen, uint8_t *response, uint8_t &responseLen)=0

  Send an APDU command to the card and receive the response.
- virtual bool sendAPDULarge (const uint8_t *apdu, uint8_t apduLen, uint8_t *response, uint16_t &responseLen)

  Send an APDU and receive a response that may exceed 255 bytes.
- virtual void resetReader ()=0

  Reset the NFC reader/field for the next card detection cycle.
- virtual bool printFirmwareVersion ()=0

  Print NFC module firmware version information to the logger.
- virtual ~CW_NfcTransport ()

3.6.1 Detailed Description

Abstract interface for NFC transport operations.
Defines the hardware-agnostic contract for NFC communication so that CW_SecureChannel and CryptnoxWallet remain independent of the physical NFC module (PN532, PN7150, etc.).
Definition at line 40 of file CW_NfcTransport.h.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 ~CW_NfcTransport()

```
virtual CW_NfcTransport::~CW_NfcTransport ()  [inline], [virtual]
```

Definition at line 103 of file CW_NfcTransport.h.

3.6.3 Member Function Documentation

3.6.3.1 begin()

```
virtual bool CW_NfcTransport::begin ()  [pure virtual]
```

Initialize the NFC transport hardware.

Returns

true if initialization succeeded, false otherwise.

3.6.3.2 inListPassiveTarget()

```
virtual bool CW_NfcTransport::inListPassiveTarget ()  [pure virtual]
```

Detect the presence of a passive ISO-DEP NFC target.

Returns

true if a card is detected, false otherwise.

3.6.3.3 printFirmwareVersion()

```
virtual bool CW_NfcTransport::printFirmwareVersion ()  [pure virtual]
```

Print NFC module firmware version information to the logger.

Returns

true if firmware info was retrieved successfully, false otherwise.

3.6.3.4 resetReader()

```
virtual void CW_NfcTransport::resetReader ()  [pure virtual]
```

Reset the NFC reader/field for the next card detection cycle.

3.6.3.5 sendAPDU()

```
virtual bool CW_NfcTransport::sendAPDU (
            const uint8_t * apdu,
            uint8_t apduLen,
            uint8_t * response,
            uint8_t & responseLen)  [pure virtual]
```

Send an APDU command to the card and receive the response.

Parameters

`in`	*apdu*	APDU command bytes.
`in`	*apduLen*	Length of the APDU command.
`out`	*response*	Buffer to receive the card response.
`out`	*responseLen*	Actual number of bytes written to `response`.

Returns

true if the exchange succeeded, false otherwise.

Referenced by sendAPDULarge().

3.6.3.6 sendAPDULarge()

```
virtual bool CW_NfcTransport::sendAPDULarge (
            const uint8_t * apdu,
            uint8_t apduLen,
            uint8_t * response,
            uint16_t & responseLen)  [inline], [virtual]
```

Send an APDU and receive a response that may exceed 255 bytes.
Used for APDUs whose DataOut can be larger than a uint8_t can express (e.g. GET_↩ MANUFACTURER_CERTIFICATE returns up to 415 bytes). Implementations that cannot deliver more than 255 bytes may delegate to sendAPDU; the default below does exactly that.

Parameters

in	*apdu*	APDU command bytes.
in	*apduLen*	Length of the APDU command.
out	*response*	Buffer to receive the card response.
in,out	*responseLen*	On entry: capacity of response. On exit: actual bytes written.

Returns

true if the exchange succeeded, false otherwise.

Definition at line 81 of file CW_NfcTransport.h.
References sendAPDU().
The documentation for this class was generated from the following file:

- CW_NfcTransport.h

3.7 CW_Platform Class Reference

Abstract interface for platform-specific operations used by the SDK.
`#include <CW_Platform.h>`

Public Member Functions

- virtual void sleep_ms (uint32_t ms)=0
  Block for at least ms milliseconds.
- virtual ~CW_Platform ()

3.7.1 Detailed Description

Abstract interface for platform-specific operations used by the SDK.
Decouples the SDK core from any specific RTOS or bare-metal delay mechanism, allowing the same SDK to run on ESP32 (FreeRTOS), Arduino, and hosted (Linux/macOS) test environments.
Definition at line 39 of file CW_Platform.h.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 ~CW_Platform()

```
virtual CW_Platform::~CW_Platform ()  [inline], [virtual]
```

Definition at line 48 of file CW_Platform.h.

3.7.3 Member Function Documentation

3.7.3.1 sleep_ms()

```
virtual void CW_Platform::sleep_ms (
            uint32_t ms)  [pure virtual]
```

Block for at least `ms` milliseconds.

Parameters

`in`	*ms*	Duration to sleep in milliseconds.

The documentation for this class was generated from the following file:

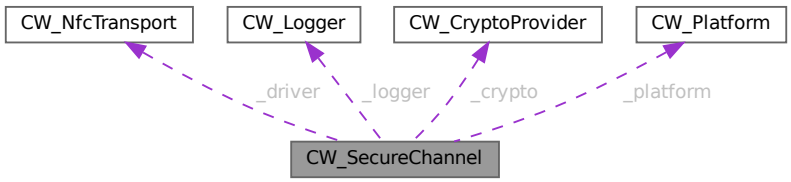
- CW_Platform.h

3.8 CW_SecureChannel Class Reference

Implements the Cryptnox secure channel protocol over NFC.

`#include <CW_SecureChannel.h>`

Collaboration diagram for CW_SecureChannel:



Public Member Functions

- CW_SecureChannel (CW_NfcTransport &driver, CW_Logger &logger, CW_CryptoProvider &crypto, CW_Platform &platform)

  Construct a CW_SecureChannel.
- CW_SecureChannel (const CW_SecureChannel &)=delete
- CW_SecureChannel & operator= (const CW_SecureChannel &)=delete
- bool begin ()

  Initialize the NFC transport module.
- bool inListPassiveTarget ()

  Detect a passive NFC target (ISO-DEP card).
- void resetReader ()

  Reset the NFC reader hardware.
- bool printFirmwareVersion ()

  Print the NFC reader firmware version to the logger.
- bool selectApdu ()

  Send the SELECT APDU to activate the Cryptnox application.
- bool getCardCertificate (uint8_t *cardCertificate, uint8_t &cardCertificateLength)

  Retrieve the card's ephemeral public key via GET CARD CERTIFICATE.
- bool extractCardEphemeralKey (const uint8_t *cardCertificate, uint8_t *cardEphemeralPubKey, uint8_t *fullEphemeralPubKey65=NULL)

Extract the card's ephemeral EC P-256 public key from a certificate.

- bool openSecureChannel (uint8_t *salt, uint8_t *clientPublicKey, uint8_t *clientPrivateKey, CW_Curve sessionCurve)

  Send OPEN SECURE CHANNEL and retrieve the session salt.
- bool mutuallyAuthenticate (CW_SecureSession &session, const uint8_t *salt, uint8_t *clientPublicKey, const uint8_t *clientPrivateKey, CW_Curve sessionCurve, const uint8_t *cardEphemeralPubKey)

  Perform ECDH derivation and MUTUALLY AUTHENTICATE with the card.
- bool getManufacturerCertificate (uint8_t *cert, uint16_t &certLen)

  Retrieve the manufacturer certificate stored in card flash.
- bool preFetchManufacturerCert ()

  Fetch and cache the manufacturer certificate before getCardCertificate().
- uint8_t verifyCertificateChain (const uint8_t *cardCert, uint8_t cardCertLen)

  Verify the full card certificate chain against the trusted CA.
- bool aesCbcEncrypt (CW_SecureSession &session, const uint8_t apdu[], uint16_t apduLength, const uint8_t data[], uint16_t dataLength, uint8_t *decryptedOutput=NULL, uint16_t *decryptedOutputLength=NULL)

  AES-CBC encrypt + MAC, send APDU, and decrypt response.
- bool aesCbcDecrypt (const CW_SecureSession &session, uint8_t *response, size_t responseLen, uint8_t *macValue, uint8_t *decryptedOutput=NULL, uint16_t *decryptedOutputLength=NULL)

  Verify MAC and decrypt an encrypted APDU response.
- bool checkStatusWord (const uint8_t *response, uint16_t responseLength, uint8_t sw1Expected, uint8_t sw2Expected)

  Verify the SW1/SW2 status word at the end of an APDU response.

Private Member Functions

- bool verifyEcdsaSha256 (const uint8_t *pubKey64, const uint8_t *message, uint16_t msgLen, const uint8_t *derSig, uint8_t derSigLen)

Static Private Member Functions

- static bool parseDerSigToRaw (const uint8_t *der, uint8_t derLen, uint8_t *raw64)

Private Attributes

- CW_NfcTransport & _driver

  NFC transport for APDU exchange.
- CW_Logger & _logger

  Logging interface.
- CW_CryptoProvider & _crypto

  Crypto operations (AES, SHA, ECDH, RNG).
- CW_Platform & _platform

  Platform abstraction (sleep_ms).
- uint8_t _lastNonce [CW_CERT_NONCE_SIZE]

  Nonce sent in the last getCardCertificate() call; checked in verifyCertificateChain().
- uint16_t _cachedMfCertLen

  Non-zero when s_mfCertBuf holds a valid pre-fetched manufacturer certificate.

3.8.1 Detailed Description

Implements the Cryptnox secure channel protocol over NFC.
Handles all low-level APDU exchanges required to establish and use a secure session with the Cryptnox smart card:

- Application selection (SELECT APDU)
- Card certificate retrieval and ephemeral key extraction
- ECDH-based session key derivation (OPEN SECURE CHANNEL + MUTUALLY AUTHENTICATE)
- AES-CBC-MAC secure messaging (encrypt / decrypt / MAC verify)
- Status word checking

CW_SecureChannel is composed inside CryptnoxWallet and is not intended to be used directly by application code.
Definition at line 64 of file CW_SecureChannel.h.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 CW_SecureChannel() [1/2]

```
CW_SecureChannel::CW_SecureChannel (
            CW_NfcTransport & driver,
            CW_Logger & logger,
            CW_CryptoProvider & crypto,
            CW_Platform & platform)
```

Construct a CW_SecureChannel.

Parameters

driver	Reference to the NFC transport.
logger	Reference to the logging interface.
crypto	Reference to the crypto provider.
platform	Reference to the platform abstraction (for sleep_ms).

Definition at line 87 of file CW_SecureChannel.cpp.
References _cachedMfCertLen, _crypto, _driver, _lastNonce, _logger, and _platform.
Referenced by CW_SecureChannel(), and operator=().

3.8.2.2 CW_SecureChannel() [2/2]

```
CW_SecureChannel::CW_SecureChannel (
            const CW_SecureChannel & )  [delete]
```

References CW_SecureChannel().

3.8.3 Member Function Documentation

3.8.3.1 aesCbcDecrypt()

```
bool CW_SecureChannel::aesCbcDecrypt (
            const CW_SecureSession & session,
            uint8_t * response,
            size_t responseLen,
            uint8_t * macValue,
            uint8_t * decryptedOutput = NULL,
            uint16_t * decryptedOutputLength = NULL)
```

Verify MAC and decrypt an encrypted APDU response.

Internal helper called from aesCbcEncrypt — exposed for the fuzz harness. Verifies the response MAC against Kmac, then decrypts the payload with Kenc using the supplied IV (which is the MAC of the sent request, by protocol).

Parameters

in,out	*session*	Secure session.
in	*response*	Encrypted response buffer (MAC || cipher || SW).
in	*responseLen*	Response length.
in	*macValue*	MAC of the request — used as decrypt IV.
out	*decryptedOutput*	Optional plaintext output buffer.
out	*decryptedOutputLength*	Optional plaintext output length.

Returns

true if MAC matched and decryption succeeded, false otherwise.

Warning

A false return indicates either tampering or a corrupted channel — the session must not be reused without renegotiation.

Definition at line 625 of file CW_SecureChannel.cpp.
References _crypto, _logger, AES_BLOCK_SIZE, CW_SecureSession::aesKey, F, HEX, CW_SecureSession::macKey, s_apduBuf, s_dataBuf, s_macBuf, CW_Utils::safe_memcpy(), CW_Utils::secure_compare(), and CW_Utils::secure_wipe().
Referenced by aesCbcEncrypt().

3.8.3.2 aesCbcEncrypt()

```
bool CW_SecureChannel::aesCbcEncrypt (
            CW_SecureSession & session,
            const uint8_t apdu[],
            uint16_t apduLength,
            const uint8_t data[],
            uint16_t dataLength,
            uint8_t * decryptedOutput = NULL,
            uint16_t * decryptedOutputLength = NULL)
```

AES-CBC encrypt + MAC, send APDU, and decrypt response.
Performs one secure messaging round-trip: pads and encrypts `data` with Kenc using the current IV; computes a CMAC over (header || cipher) with Kmac; sends the wrapped APDU; on the response, verifies the MAC and decrypts the payload. The new IV for the next call is taken from the last cipher block (rolling IV).

Parameters

in,out	*session*	Secure session (Kenc / Kmac / IV).
in	*apdu*	APDU header (CLA, INS, P1, P2).
in	*apduLength*	Header length (must be 4).
in	*data*	Plaintext payload.
in	*dataLength*	Plaintext length ($\leq$ CW_USER_DATA_PAGE_SIZE).
out	*decryptedOutput*	Optional buffer for the decrypted response payload.
out	*decryptedOutputLength*	Optional pointer to receive the decrypted payload length.

Returns

true if the APDU was sent and the response MAC verified + decrypted successfully; false on bad parameters, transport failure, MAC mismatch, or unexpected status word.

Precondition

`session` must be the output of a successful mutuallyAuthenticate call.

Warning

Mutates `session.iv`. On any failure the IV may be left in an undefined state — treat the session as broken and tear it down with CW_SecureSession::clear.

Reuses module-private static buffers (`s_apduBuf`, `s_macBuf`, `s_dataBuf`). Not safe to call concurrently from multiple tasks; serialise at the application level.

One secure-messaging round-trip is built in five stages, reusing module- private scratch buffers (`s_↩ apduBuf`, `s_macBuf`, `s_dataBuf`) to keep the call-site stack frame small:

1. Plaintext padding — ISO/IEC 9797-1 Method 2 (bit padding) is appended to `data` so the length is a multiple of the AES block size.
2. Encryption — AES-256-CBC under Kenc with the current rolling IV.
3. MAC computation — AES-CMAC over (APDU header || Lc || ciphertext) under Kmac. The MAC's last 8 bytes are prepended to the ciphertext in the outgoing APDU.
4. Transport — the assembled APDU goes to the card; the response is structured as (MAC[8] || cipher || SW1 SW2).
5. Response decryption — delegated to aesCbcDecrypt, which verifies the response MAC against Kmac (using the request MAC as the IV per protocol) before decrypting under Kenc.

IV update — on success, the last 16 bytes of the response ciphertext become the new session IV (rolling IV). On any failure path the IV is left in an undefined state, which is why the caller must treat a false return as a dead session.
Definition at line 501 of file CW_SecureChannel.cpp.
References _crypto, _driver, _logger, AES_BLOCK_SIZE, aesCbcDecrypt(), CW_SecureSession::aesKey, APDU_LC_LEN, checkStatusWord(), CW_SecureSession::clear(), CW_IV_SIZE, F, HEX, INPUT_BUFFER_LIMIT, CW_SecureSession::iv, MAC_APDU_LEN, CW_SecureSession::macKey, MAX_MAC_DATA_LEN, s_apduBuf, s_dataBuf, s_macBuf, CW_Utils::safe_memcpy(), CW_Utils::secure_wipe(), and SEND_APDU_MAX_LEN.

3.8.3.3 begin()

```
bool CW_SecureChannel::begin ()
```

Initialize the NFC transport module.

Returns

true if the module was successfully initialised, false otherwise.

Definition at line 100 of file CW_SecureChannel.cpp.
References _driver.

3.8.3.4 checkStatusWord()

```
bool CW_SecureChannel::checkStatusWord (
            const uint8_t * response,
            uint16_t responseLength,
            uint8_t sw1Expected,
            uint8_t sw2Expected)
```

Verify the SW1/SW2 status word at the end of an APDU response.

Parameters

response	APDU response buffer.
responseLength	Response length.
sw1Expected	Expected SW1 byte.
sw2Expected	Expected SW2 byte.

Returns

true if last two bytes match expectations, false otherwise.

Definition at line 120 of file CW_SecureChannel.cpp.
References _logger, F, and HEX.
Referenced by aesCbcEncrypt(), getCardCertificate(), getManufacturerCertificate(), mutuallyAuthenticate(), openSecureChannel(), and selectApdu().

3.8.3.5 extractCardEphemeralKey()

```
bool CW_SecureChannel::extractCardEphemeralKey (
            const uint8_t * cardCertificate,
            uint8_t * cardEphemeralPubKey,
            uint8_t * fullEphemeralPubKey65 = NULL)
```

Extract the card's ephemeral EC P-256 public key from a certificate.

Parameters

in	*cardCertificate*	Raw certificate bytes.
out	*cardEphemeralPubKey*	64-byte key (X‖Y, no 0x04 prefix) for ECDH.
out	*fullEphemeralPubKey65*	Optional 65-byte key including 0x04 prefix.

Returns

true on success, false otherwise.

Definition at line 232 of file CW_SecureChannel.cpp.

3.8.3.6 getCardCertificate()

```
bool CW_SecureChannel::getCardCertificate (
            uint8_t * cardCertificate,
            uint8_t & cardCertificateLength)
```

Retrieve the card's ephemeral public key via GET CARD CERTIFICATE.
Sends a random challenge nonce to the card and stores it internally. The nonce echo check is performed inside verifyCertificateChain() to ensure replay protection is coupled with signature verification.

Parameters

out	*cardCertificate*	Buffer to receive the raw certificate bytes.
out	*cardCertificateLength*	Actual certificate length (bytes).

Returns

true on success, false otherwise.

Definition at line 184 of file CW_SecureChannel.cpp.
References _crypto, _driver, _lastNonce, _logger, checkStatusWord(), F, GETCARDCERTIFICATE_IN_BYTES, RANDOM_BYTES, RESPONSE_GETCARDCERTIFICATE_IN_BYTES, RESPONSE_STATUS_WORDS_IN_BYTES, and CW_Utils::safe_memcpy().

3.8.3.7 getManufacturerCertificate()

```
bool CW_SecureChannel::getManufacturerCertificate (
            uint8_t * cert,
            uint16_t & certLen)
```

Retrieve the manufacturer certificate stored in card flash.

Parameters

out	*cert*	Buffer to receive the raw DER certificate bytes.
out	*certLen*	Actual certificate length written.

Returns

true on success, false if APDU fails or buffer too small.

Definition at line 1098 of file CW_SecureChannel.cpp.
References _driver, _logger, checkStatusWord(), CW_MANUF_CERT_MAX_BYTES, F, RESPONSE_GETMANUFACTURERCERT_PAGE_IN_BYTES, RESPONSE_STATUS_WORDS_IN_BYTES, and CW_Utils::safe_memcpy().
Referenced by preFetchManufacturerCert(), and verifyCertificateChain().

3.8.3.8 inListPassiveTarget()

```
bool CW_SecureChannel::inListPassiveTarget ()
```

Detect a passive NFC target (ISO-DEP card).

Returns

true if a card was found, false otherwise.

Definition at line 104 of file CW_SecureChannel.cpp.
References _driver.

3.8.3.9 mutuallyAuthenticate()

```
bool CW_SecureChannel::mutuallyAuthenticate (
            CW_SecureSession & session,
            const uint8_t * salt,
            uint8_t * clientPublicKey,
            const uint8_t * clientPrivateKey,
            CW_Curve sessionCurve,
            const uint8_t * cardEphemeralPubKey)
```

Perform ECDH derivation and MUTUALLY AUTHENTICATE with the card.
Final step of the secure channel handshake:

1. ECDH shared secret = clientPrivateKey cardEphemeralPubKey
2. (Kenc || Kmac || IV) ← SHA-512(salt || pairingKey || sharedSecret)
3. Encrypts a 16-byte random challenge with the new Kenc and sends it inside the MUTUALLY AUTHENTICATE APDU
4. Verifies the card returns the same plaintext when re-encrypting its own counter — this proves the card knows Kenc

Parameters

out	*session*	Secure session populated with derived keys + initial IV.
in	*salt*	32-byte salt from openSecureChannel.
in	*clientPublicKey*	64-byte client public key.
in	*clientPrivateKey*	32-byte client private key.
in	*sessionCurve*	ECC curve.
in	*cardEphemeralPubKey*	64-byte card ephemeral public key.

Returns

true on success, false if ECDH failed, the card's challenge response did not match, or any APDU exchange failed.

Precondition

openSecureChannel must have been called and returned true.

Postcondition

On true: `session` has Kenc, Kmac, and rolling IV ready for aesCbcEncrypt. On false: `session` is left untouched and must not be used.

Warning

All ephemeral key material in the caller's stack (`clientPrivateKey`, `salt`) must be wiped after this call.

Cryptographic flow:

1. Compute the ECDH shared secret S = ECDH(clientPrivateKey, cardEphemeralPubKey) on the negotiated curve.
2. Derive 80 bytes of keying material via SHA-512 over (salt || S || pairingDataHash) and split into:
   - Kenc[32]: AES-256 session encryption key
   - Kmac[32]: AES-256 session MAC key
   - IV[16] : initial rolling IV
3. Send MUTUALLY AUTHENTICATE with a random 16-byte challenge encrypted under Kenc / IV. The card must answer with the same 16 bytes re-encrypted with the new IV — a verification that fails fast on any key-derivation mismatch.
4. On success, populate `session` and wipe `sharedSecret` from the stack.

Failure modes that cause an early-exit with a wiped session:

- ECDH returned zero or invalid (curve mismatch)
- APDU transport failure
- Card challenge response mismatch (active attacker or wrong card)

Definition at line 334 of file CW_SecureChannel.cpp.
References _crypto, _driver, _logger, AES_BLOCK_SIZE, CW_SecureSession::aesKey, APDU_HEADER_LEN, APDU_LC_LEN, checkStatusWord(), CW_SecureSession::clear(), COMMON_PAIRING_DATA, CW_AESKEY_SIZE, CW_IV_SIZE, CW_MACKEY_SIZE, F, CW_SecureSession::iv, CW_SecureSession::macKey, REQUEST_MUTUALLYAUTHENTICATE_IN_BYTES, RESPONSE_MUTUALLYAUTHENTICATE_IN_BYTES, CW_Utils::safe_memcpy(), and CW_Utils::secure_wipe().

3.8.3.10 openSecureChannel()

```
bool CW_SecureChannel::openSecureChannel (
            uint8_t * salt,
            uint8_t * clientPublicKey,
            uint8_t * clientPrivateKey,
            CW_Curve sessionCurve)
```

Send OPEN SECURE CHANNEL and retrieve the session salt.
Generates a client EC key pair via the crypto provider and sends the public key to the card; the card responds with a 32-byte salt that later feeds into Kenc / Kmac derivation in mutuallyAuthenticate.

Parameters

out	*salt*	32-byte session salt returned by the card.
out	*clientPublicKey*	64-byte freshly generated client public key.
out	*clientPrivateKey*	32-byte freshly generated client private key.
in	*sessionCurve*	ECC curve for key generation (secp256r1).

Returns

true on success, false otherwise.

Precondition

selectApdu must have been called successfully.

Warning

`clientPrivateKey` is sensitive — the caller MUST wipe it via CW_Utils::secure_wipe on every exit path.

Definition at line 265 of file CW_SecureChannel.cpp.
References _crypto, _driver, _logger, checkStatusWord(), CLIENT_PUBLIC_KEY_SIZE, F, OPENSECURECHANNEL_SALT_IN_BYTES, RESPONSE_OPENSECURECHANNEL_IN_BYTES, and CW_Utils::safe_memcpy().

3.8.3.11 operator=()

```
CW_SecureChannel & CW_SecureChannel::operator= (
            const CW_SecureChannel & )  [delete]
```

References CW_SecureChannel().

3.8.3.12 parseDerSigToRaw()

```
bool CW_SecureChannel::parseDerSigToRaw (
            const uint8_t * der,
            uint8_t derLen,
            uint8_t * raw64)  [static], [private]
```

Definition at line 1018 of file CW_SecureChannel.cpp.
References CW_Utils::safe_memcpy().
Referenced by verifyEcdsaSha256().

3.8.3.13 preFetchManufacturerCert()

```
bool CW_SecureChannel::preFetchManufacturerCert ()
```

Fetch and cache the manufacturer certificate before getCardCertificate().
The Cryptnox card state machine advances after GET_CARD_CERTIFICATE (INS=F8) and will not respond to GET_MANUFACTURER_CERTIFICATE (INS=F7) after that point. Call this method immediately after selectApdu() and before getCardCertificate() so that verifyCertificateChain() can use the cached copy without an APDU.

Returns

true if the certificate was fetched and cached, false otherwise.

Definition at line 1194 of file CW_SecureChannel.cpp.
References _cachedMfCertLen, getManufacturerCertificate(), and s_mfCertBuf.

3.8.3.14 printFirmwareVersion()

```
bool CW_SecureChannel::printFirmwareVersion ()
```

Print the NFC reader firmware version to the logger.

Returns

true on success, false otherwise.

Definition at line 112 of file CW_SecureChannel.cpp.
References _driver.

3.8.3.15 resetReader()

```
void CW_SecureChannel::resetReader ()
```

Reset the NFC reader hardware.
Definition at line 108 of file CW_SecureChannel.cpp.
References _driver.

3.8.3.16 selectApdu()

```
bool CW_SecureChannel::selectApdu ()
```

Send the SELECT APDU to activate the Cryptnox application.

Returns

true on success, false otherwise.

Definition at line 155 of file CW_SecureChannel.cpp.
References _driver, _logger, checkStatusWord(), F, and RESPONSE_SELECT_IN_BYTES.

3.8.3.17 verifyCertificateChain()

```
uint8_t CW_SecureChannel::verifyCertificateChain (
            const uint8_t * cardCert,
            uint8_t cardCertLen)
```

Verify the full card certificate chain against the trusted CA.
Walks the cached manufacturer certificate (fetched earlier by preFetchManufacturerCert), verifies its ECDSA signature against each entry in CW_TRUSTED_CA_KEYS, then verifies the card's ephemeral certificate against the manufacturer public key. Also checks that the challenge nonce sent in getCardCertificate was echoed back inside the card certificate.

Parameters

in	*cardCert*	Raw card certificate bytes (typically 146 bytes).
in	*cardCertLen*	Length of `cardCert`.

Returns

One of the `CW_CERT_*` result codes:

Return values

CW_CERT_OK	Chain verified end-to-end.
CW_CERT_FORMAT_ERROR	Malformed certificate / unexpected TLV.
CW_CERT_NONCE_MISMATCH	Card did not echo the challenge nonce.
CW_CERT_CARD_SIG_INVALID	Card cert ECDSA signature failed verification.
CW_CERT_MANUF_SIG_INVALID	Manufacturer cert signature does not match any trusted CA key.
CW_CERT_KEY_NOT_FOUND	Device public-key OID not found in the certificate.

Precondition

preFetchManufacturerCert must have been called and returned true (the manufacturer certificate is cached internally and not re-fetchable after getCardCertificate).

getCardCertificate must have been called so the challenge nonce is recorded internally.

Two-step ECDSA chain walk against the pinned trusted CAs in CW_TRUSTED_CA_KEYS (currently a single secp256r1 key, CW_CA_DLT_PUBKEY):

1. Manufacturer certificate — the cached DER blob fetched by preFetchManufacturerCert is parsed to extract its EC public key and its ECDSA signature. The signature is verified against every entry in the trusted-CA table; the first match wins. The extracted manufacturer public key becomes the trusted issuer for step 2.
2. Card certificate — `cardCert` is parsed for the device's ephemeral public key, the manufacturer ECDSA signature over that key, and the echoed challenge nonce. The nonce is compared (constant-time) against the value stored by getCardCertificate; mismatch immediately returns CW_CERT_NONCE_MISMATCH to defeat replay. The signature is then verified against the manufacturer public key from step 1.

Both signatures are SHA-256-then-ECDSA over the relevant TBS bytes. Any TLV parsing error short-circuits with CW_CERT_FORMAT_ERROR rather than touching the verifier — DER parsing is the largest attack surface here and is independently fuzzed (see `fuzz/fuzz_der.cpp`).
Definition at line 1227 of file CW_SecureChannel.cpp.
References _cachedMfCertLen, _lastNonce, _logger, CW_CERT_CARD_SIG_INVALID, CW_CERT_FORMAT_ERROR, CW_CERT_KEY_NOT_FOUND, CW_CERT_MANUF_SIG_INVALID, CW_CERT_NONCE_MISMATCH, CW_CERT_NONCE_SIZE, CW_CERT_OK, CW_TRUSTED_CA_COUNT, CW_TRUSTED_CA_KEYS, derWalkMfCert(), F, getManufacturerCertificate(), s_mfCertBuf, CW_Utils::secure_compare(), CW_Utils::secure_wipe(), and verifyEcdsaSha256().

3.8.3.18 verifyEcdsaSha256()

```
bool CW_SecureChannel::verifyEcdsaSha256 (
            const uint8_t * pubKey64,
            const uint8_t * message,
            uint16_t msgLen,
            const uint8_t * derSig,
            uint8_t derSigLen)  [private]
```

Definition at line 1082 of file CW_SecureChannel.cpp.
References _crypto, CW_CURVE_SECP256R1, and parseDerSigToRaw().
Referenced by verifyCertificateChain().

3.8.4 Member Data Documentation

3.8.4.1 _cachedMfCertLen

```
uint16_t CW_SecureChannel::_cachedMfCertLen  [private]
```

Non-zero when s_mfCertBuf holds a valid pre-fetched manufacturer certificate.

Definition at line 322 of file CW_SecureChannel.h.
Referenced by CW_SecureChannel(), preFetchManufacturerCert(), and verifyCertificateChain().

3.8.4.2 _crypto

```
CW_CryptoProvider& CW_SecureChannel::_crypto  [private]
```
Crypto operations (AES, SHA, ECDH, RNG).
Definition at line 315 of file CW_SecureChannel.h.
Referenced by aesCbcDecrypt(), aesCbcEncrypt(), CW_SecureChannel(), getCardCertificate(), mutuallyAuthenticate(), openSecureChannel(), and verifyEcdsaSha256().

3.8.4.3 _driver

```
CW_NfcTransport& CW_SecureChannel::_driver  [private]
```
NFC transport for APDU exchange.
Definition at line 313 of file CW_SecureChannel.h.
Referenced by aesCbcEncrypt(), begin(), CW_SecureChannel(), getCardCertificate(), getManufacturerCertificate(), inListPassiveTarget(), mutuallyAuthenticate(), openSecureChannel(), printFirmwareVersion(), resetReader(), and selectApdu().

3.8.4.4 _lastNonce

```
uint8_t CW_SecureChannel::_lastNonce[CW_CERT_NONCE_SIZE]  [private]
```
Nonce sent in the last getCardCertificate() call; checked in verifyCertificateChain().
Definition at line 319 of file CW_SecureChannel.h.
Referenced by CW_SecureChannel(), getCardCertificate(), and verifyCertificateChain().

3.8.4.5 _logger

```
CW_Logger& CW_SecureChannel::_logger  [private]
```
Logging interface.
Definition at line 314 of file CW_SecureChannel.h.
Referenced by aesCbcDecrypt(), aesCbcEncrypt(), checkStatusWord(), CW_SecureChannel(), getCardCertificate(), getManufacturerCertificate(), mutuallyAuthenticate(), openSecureChannel(), selectApdu(), and verifyCertificateChain().

3.8.4.6 _platform

```
CW_Platform& CW_SecureChannel::_platform  [private]
```
Platform abstraction (sleep_ms).
Definition at line 316 of file CW_SecureChannel.h.
Referenced by CW_SecureChannel().
The documentation for this class was generated from the following files:

- CW_SecureChannel.h
- CW_SecureChannel.cpp

3.9 CW_SecureSession Struct Reference

Holds cryptographic session state for reentrant secure channel operations.
`#include <CW_Defs.h>`

Public Member Functions

- CW_SecureSession ()
  Zero-initialise all session keys and IV.
- void clear ()
  Securely clear all session keys and IV.

Public Attributes

- uint8_t aesKey [CW_AESKEY_SIZE]
- uint8_t macKey [CW_MACKEY_SIZE]
- uint8_t iv [CW_IV_SIZE]

3.9.1 Detailed Description

Holds cryptographic session state for reentrant secure channel operations.
Encapsulates all session-specific cryptographic material (Kenc, Kmac, rolling IV), allowing functions to be reentrant by passing session state as a parameter.
Definition at line 168 of file CW_Defs.h.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 CW_SecureSession()

```
CW_SecureSession::CW_SecureSession () [inline]
```

Zero-initialise all session keys and IV.
Definition at line 174 of file CW_Defs.h.
References aesKey, iv, and macKey.

3.9.3 Member Function Documentation

3.9.3.1 clear()

```
void CW_SecureSession::clear () [inline]
```

Securely clear all session keys and IV.
Definition at line 181 of file CW_Defs.h.
References aesKey, iv, macKey, and CW_Utils::secure_wipe().
Referenced by CW_SecureChannel::aesCbcEncrypt(), CryptnoxWallet::connect(), CryptnoxWallet::disconnect(), and CW_SecureChannel::mutuallyAuthenticate().

3.9.4 Member Data Documentation

3.9.4.1 aesKey

```
uint8_t CW_SecureSession::aesKey[CW_AESKEY_SIZE]
```

AES-256 session encryption key (Kenc)
Definition at line 169 of file CW_Defs.h.
Referenced by CW_SecureChannel::aesCbcDecrypt(), CW_SecureChannel::aesCbcEncrypt(), clear(), CW_SecureSession(), CryptnoxWallet::isSecureChannelOpen(), and CW_SecureChannel::mutuallyAuthenticate().

3.9.4.2 iv

```
uint8_t CW_SecureSession::iv[CW_IV_SIZE]
```

Current AES-CBC IV (rolling IV)
Definition at line 171 of file CW_Defs.h.
Referenced by CW_SecureChannel::aesCbcEncrypt(), clear(), CW_SecureSession(), and CW_SecureChannel::mutuallyAuthenticate().

3.9.4.3 macKey

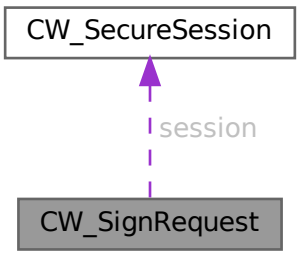
```
uint8_t CW_SecureSession::macKey[CW_MACKEY_SIZE]
```

AES-256 session MAC key (Kmac)
Definition at line 170 of file CW_Defs.h.
Referenced by CW_SecureChannel::aesCbcDecrypt(), CW_SecureChannel::aesCbcEncrypt(), clear(), CW_SecureSession(), and CW_SecureChannel::mutuallyAuthenticate().
The documentation for this struct was generated from the following file:

- CW_Defs.h

3.10 CW_SignRequest Struct Reference

Request parameters for CryptnoxWallet::sign.
`#include <CryptnoxWallet.h>`
Collaboration diagram for CW_SignRequest:



Public Member Functions

- CW_SignRequest (CW_SecureSession &sess, uint8_t kType=CW_SIGN_CURR_K1, uint8_t sigType=CW_SIGN_SIG_ECDSA_LOW_S, bool pinless=CW_SIGN_WITH_PIN)

  Construct a sign request with sensible defaults.
- ~CW_SignRequest ()

  Securely wipes the PIN buffer.

Public Attributes

- CW_SecureSession & session
- uint8_t keyType
- uint8_t signatureType
- uint8_t pin [CW_MAX_PIN_LENGTH]
- bool pinLessMode
- const uint8_t * hash
- uint8_t hashLength
- const uint8_t * derivePath
- uint8_t derivePathLength

3.10.1 Detailed Description

Request parameters for CryptnoxWallet::sign.
Owns the PIN buffer for the lifetime of the request — the destructor securely wipes it (CW_Utils::secure_wipe), so allocating the request on the stack inside a tight scope is the recommended pattern.
Definition at line 74 of file CryptnoxWallet.h.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 CW_SignRequest()

```
CW_SignRequest::CW_SignRequest (
            CW_SecureSession & sess,
            uint8_t kType = CW_SIGN_CURR_K1,
            uint8_t sigType = CW_SIGN_SIG_ECDSA_LOW_S,
            bool pinless = CW_SIGN_WITH_PIN)  [inline], [explicit]
```

Construct a sign request with sensible defaults.

Parameters

in	*sess*	Open secure session.
in	*kType*	Key type. Defaults to CW_SIGN_CURR_K1.
in	*sigType*	Signature type. Defaults to CW_SIGN_SIG_ECDSA_LOW_S.
in	*pinless*	PIN mode. Defaults to PIN required (CW_SIGN_WITH_PIN).

Definition at line 92 of file CryptnoxWallet.h.
References CW_SIGN_CURR_K1, CW_SIGN_SIG_ECDSA_LOW_S, CW_SIGN_WITH_PIN, derivePath, derivePathLength, hash, hashLength, keyType, pin, pinLessMode, session, and signatureType.

3.10.2.2 ~CW_SignRequest()

```
CW_SignRequest::~CW_SignRequest ()  [inline]
```

Securely wipes the PIN buffer.
Definition at line 103 of file CryptnoxWallet.h.
References pin, and CW_Utils::secure_wipe().

3.10.3 Member Data Documentation

3.10.3.1 derivePath

```
const uint8_t* CW_SignRequest::derivePath
```

BIP32 path bytes for DERIVE modes; `NULL` for CURR / PINLESS modes.
Definition at line 82 of file CryptnoxWallet.h.
Referenced by CryptnoxWallet::buildSignPayload(), and CW_SignRequest().

3.10.3.2 derivePathLength

```
uint8_t CW_SignRequest::derivePathLength
```

Length of `derivePath` in bytes (must be a multiple of 4).
Definition at line 83 of file CryptnoxWallet.h.
Referenced by CryptnoxWallet::buildSignPayload(), and CW_SignRequest().

3.10.3.3 hash

```
const uint8_t* CW_SignRequest::hash
```

Pointer to the hash to sign (typically 32 bytes — SHA-256 of the transaction).
Definition at line 80 of file CryptnoxWallet.h.
Referenced by CryptnoxWallet::buildSignPayload(), CW_SignRequest(), and CryptnoxWallet::validateSignRequest().

3.10.3.4 hashLength

```
uint8_t CW_SignRequest::hashLength
```

Length of `hash` in bytes (must be $\leq$ CW_HASH_SIZE).
Definition at line 81 of file CryptnoxWallet.h.

Referenced by CryptnoxWallet::buildSignPayload(), CW_SignRequest(), and CryptnoxWallet::validateSignRequest().

3.10.3.5 keyType

`uint8_t CW_SignRequest::keyType`
Key / path type — one of the `CW_SIGN_CURR_*`, `CW_SIGN_DERIVE_*`, `CW_SIGN_PINLESS_K1` constants.
Definition at line 76 of file CryptnoxWallet.h.
Referenced by CryptnoxWallet::buildSignPayload(), CW_SignRequest(), CryptnoxWallet::sendSignApdu(), and CryptnoxWallet::validateSignRequest().

3.10.3.6 pin

`uint8_t CW_SignRequest::pin[CW_MAX_PIN_LENGTH]`
PIN bytes (4–9 ASCII digits). Zero-padded; cleared in the destructor.
Definition at line 78 of file CryptnoxWallet.h.
Referenced by CryptnoxWallet::buildSignPayload(), CW_SignRequest(), CryptnoxWallet::validateSignRequest(), and ~CW_SignRequest().

3.10.3.7 pinLessMode

`bool CW_SignRequest::pinLessMode`
false = PIN path, true = PIN-less path (requires `keyType` == CW_SIGN_PINLESS_K1).
Definition at line 79 of file CryptnoxWallet.h.
Referenced by CryptnoxWallet::buildSignPayload(), CW_SignRequest(), and CryptnoxWallet::validateSignRequest().

3.10.3.8 session

`CW_SecureSession& CW_SignRequest::session`
Reference to an open secure session.
Definition at line 75 of file CryptnoxWallet.h.
Referenced by CW_SignRequest(), CryptnoxWallet::sendSignApdu(), and CryptnoxWallet::validateSignRequest().

3.10.3.9 signatureType

`uint8_t CW_SignRequest::signatureType`
Signature format — one of `CW_SIGN_SIG_ECDSA_LOW_S`, `CW_SIGN_SIG_ECDSA_EOSIO`, `CW_SIGN_SIG_SCHNORR_BIP340`.
Definition at line 77 of file CryptnoxWallet.h.
Referenced by CW_SignRequest(), and CryptnoxWallet::sendSignApdu().
The documentation for this struct was generated from the following file:

- CryptnoxWallet.h

3.11 CW_SignResult Struct Reference

Result of CryptnoxWallet::sign.
`#include <CryptnoxWallet.h>`

Public Member Functions

- CW_SignResult ()
  Construct a default-failure result.

Public Attributes

- uint8_t signature [CW_RAW_SIGNATURE_SIZE]
- uint8_t errorCode

3.11.1 Detailed Description

Result of CryptnoxWallet::sign.
The error code is checked first: when it is CW_OK the signature is valid raw (r || s) on secp256k1 / secp256r1 (depending on the `keyType` used). On any other code `signature` is left zero.
Definition at line 116 of file CryptnoxWallet.h.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 CW_SignResult()

`CW_SignResult::CW_SignResult () [inline]`
Construct a default-failure result.
Definition at line 121 of file CryptnoxWallet.h.
References CW_NOK, errorCode, and signature.

3.11.3 Member Data Documentation

3.11.3.1 errorCode

`uint8_t CW_SignResult::errorCode`
CW_OK on success, otherwise a `CW_SIGN_*` / `CW_INVALID_SESSION` code.
Definition at line 118 of file CryptnoxWallet.h.
Referenced by CW_SignResult(), CryptnoxWallet::extractRawSignature(), CryptnoxWallet::sendSignApdu(), CryptnoxWallet::sign(), and CryptnoxWallet::validateSignRequest().

3.11.3.2 signature

`uint8_t CW_SignResult::signature[CW_RAW_SIGNATURE_SIZE]`
Raw 64-byte signature: r[32] || s[32]. Zero on failure.
Definition at line 117 of file CryptnoxWallet.h.
Referenced by CW_SignResult(), CryptnoxWallet::extractRawSignature(), and CryptnoxWallet::sign().
The documentation for this struct was generated from the following file:

- CryptnoxWallet.h

3.12 CW_Utils Class Reference

Portable utility functions for cryptographic and security operations.
`#include <CW_Utils.h>`

Static Public Member Functions

- static bool secure_compare (const uint8_t *a, const uint8_t *b, size_t len)
  Constant-time buffer comparison, resistant to timing side-channel attacks.
- static void secure_wipe (uint8_t *buf, size_t len)
  Securely zero a buffer, guaranteed not to be optimised away.
- static bool safe_memcpy (uint8_t *dst, size_t dstSize, const uint8_t *src, size_t count)
  Safe memcpy — validates pointers, sizes, and checks for overlap.
- static bool fill_secure_random (uint8_t *dest, size_t len)
  Fill `len` bytes at `dest` with cryptographically random data.

3.12.1 Detailed Description

Portable utility functions for cryptographic and security operations.
All methods here are platform-independent pure C++ with no dependency on Arduino or any hardware-specific library.

Hardware-specific helpers (e.g. TRNG byte generation) live in the concrete crypto provider implementation (ArduinoCryptoProvider).
Definition at line 45 of file CW_Utils.h.

3.12.2 Member Function Documentation

3.12.2.1 fill_secure_random()

```
bool CW_Utils::fill_secure_random (
            uint8_t * dest,
            size_t len)  [static]
```

Fill `len` bytes at `dest` with cryptographically random data.
The implementation is platform-specific. On ESP32 it calls esp_fill_random() after verifying that Wi-Fi or Bluetooth is active, ensuring the hardware TRNG is properly seeded. Returns false (hard failure) if neither radio is on (SEC-001).

Parameters

dest	Destination buffer.
len	Number of random bytes to generate.

Returns

true on success, false if dest is NULL, len is zero, or no radio is active.

3.12.2.2 safe_memcpy()

```
bool CW_Utils::safe_memcpy (
            uint8_t * dst,
            size_t dstSize,
            const uint8_t * src,
            size_t count)  [static]
```

Safe memcpy — validates pointers, sizes, and checks for overlap.
Safe memcpy — checks src, dst and size before copying.

Parameters

dst	Destination buffer.
dstSize	Capacity of the destination buffer.
src	Source buffer.
count	Number of bytes to copy.

Returns

true if copy succeeded, false if parameters are invalid.

true if copy succeeded, false otherwise.

Definition at line 50 of file CW_Utils.cpp.
Referenced by CW_SecureChannel::aesCbcDecrypt(), CW_SecureChannel::aesCbcEncrypt(), CryptnoxWallet::buildSignPayload(), CryptnoxWallet::extractRawSignature(), CW_SecureChannel::getCardCertificate(), CryptnoxWallet::getCardInfo(), CW_SecureChannel::getManufacturerCertificate(), CW_SecureChannel::mutuallyAuther CW_SecureChannel::openSecureChannel(), CryptnoxWallet::parseDerSignature(), CW_SecureChannel::parseDerSigT and CryptnoxWallet::verifyPin().

3.12.2.3 secure_compare()

```
bool CW_Utils::secure_compare (
            const uint8_t * a,
            const uint8_t * b,
            size_t len)  [static]
```

Constant-time buffer comparison, resistant to timing side-channel attacks.
Always iterates over the full length regardless of where the first difference occurs, preventing an attacker from inferring the correct value byte-by-byte via timing measurements.

Parameters

a	Pointer to the first buffer.
b	Pointer to the second buffer.
len	Number of bytes to compare.

Returns

true if the buffers are identical, false otherwise.

Definition at line 22 of file CW_Utils.cpp.
Referenced by CW_SecureChannel::aesCbcDecrypt(), and CW_SecureChannel::verifyCertificateChain().

3.12.2.4 secure_wipe()

```
void CW_Utils::secure_wipe (
            uint8_t * buf,
            size_t len)  [static]
```

Securely zero a buffer, guaranteed not to be optimised away.
Uses a volatile pointer so the compiler cannot elide the writes, ensuring sensitive material is actually erased from memory.

Parameters

buf	Pointer to the buffer to wipe.
len	Number of bytes to zero.

Definition at line 37 of file CW_Utils.cpp.
Referenced by CW_SecureChannel::aesCbcDecrypt(), CW_SecureChannel::aesCbcEncrypt(), CW_SecureSession::clear(), CryptnoxWallet::establishSecureChannel(), CryptnoxWallet::extractRawSignature(), CryptnoxWallet::getCardInfo(), CW_SecureChannel::mutuallyAuthenticate(), CryptnoxWallet::sign(), CW_SecureChannel::verifyCertificateChain(), CryptnoxWallet::verifyPin(), and CW_SignRequest::~CW_SignRequest().
The documentation for this class was generated from the following files:

- CW_Utils.h
- CW_Utils.cpp

Chapter 4

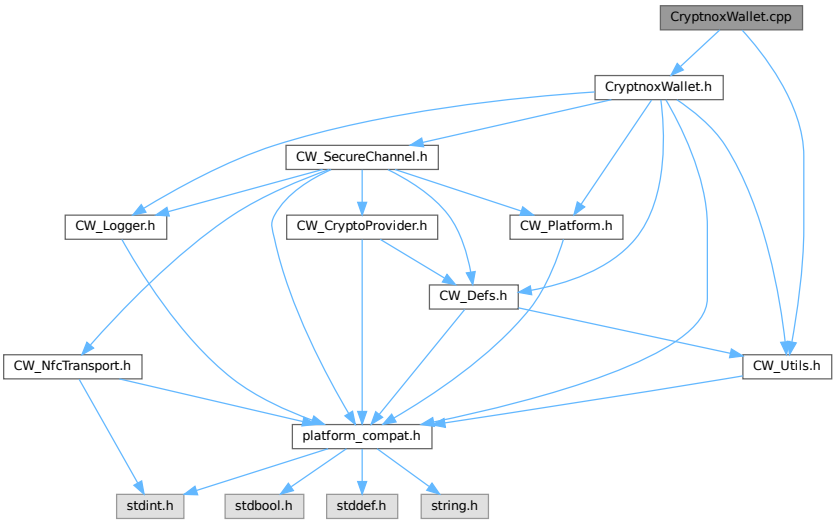
File Documentation

4.1 CryptnoxWallet.cpp File Reference

Implementation of the high-level CryptnoxWallet API.

```
#include "CryptnoxWallet.h"
#include "CW_Utils.h"
```

Include dependency graph for CryptnoxWallet.cpp:



4.1.1 Detailed Description

Implementation of the high-level CryptnoxWallet API.
Coordinates the secure channel layer (CW_SecureChannel) with the higher-level wallet operations declared in CryptnoxWallet.h. Handles connection retries, sensitive buffer wiping on every exit path, PIN/sign payload assembly, and DER signature parsing.
Definition in file CryptnoxWallet.cpp.

4.2 CryptnoxWallet.cpp

Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
```

```
00004  */
00005
00015
00016 /* NOTE: Do NOT include <Arduino.h> here -- this is a platform-independent file.
00017  * Arduino compatibility shims (F(), HEX, delay) are provided via
00018  * platform_compat.h which is pulled in transitively through CryptnoxWallet.h. */
00019 #include "CryptnoxWallet.h"
00020 #include "CW_Utils.h"
00021
00022 /**************************************************************************
00023  * Constructor
00024  **************************************************************************/
00025
00026 // cppcheck-suppress misra-c2012-12.3 -- C++: member initializer-list commas are not the comma
      operator
00027 CryptnoxWallet::CryptnoxWallet(CW_NfcTransport& driver, CW_Logger& logger,
00028                                CW_CryptoProvider& crypto, CW_Platform& platform)
00029     : _logger(logger), _platform(platform), _secure(driver, logger, crypto, platform) {
00030 }
00031
00032 /**************************************************************************
00033  * Public methods
00034  **************************************************************************/
00035
00036 bool CryptnoxWallet::begin() {
00037     bool ret = _secure.begin();
00038     if (ret) {
00039         printPN532FirmwareVersion();
00040     }
00041     return ret;
00042 }
00043
00044 bool CryptnoxWallet::connect(CW_SecureSession& session) {
00045     bool ret = false;
00046     session.clear(); /* CRIT-04: clear any stale keys from a previous or partial session */
00047
00048     for (uint8_t attempt = 0U; (attempt < CW_CONNECT_MAX_ATTEMPTS) && (ret == false); attempt++) {
00049         if (attempt > 0U) {
00050             session.clear(); /* CRIT-04: clear partial keys left by a failed attempt before retrying
      */
00051 #if CW_DEBUG_LOGGING
00052             _logger.print(F("Retrying card connection (attempt "));
00053             _logger.print((uint8_t)(attempt + 1U));
00054             _logger.println(F(")..."));
00055 #endif
00056             _secure.resetReader();
00057             _platform.sleep_ms(200U);
00058         }
00059
00060         if (_secure.inListPassiveTarget()) {
00061             _platform.sleep_ms(200U);
00062             if (establishSecureChannel(session)) {
00063                 ret = true;
00064             }
00065         }
00066     }
00067
00068     if (!ret) {
00069         session.clear(); /* CRIT-04: clear any partial keys from the final failed attempt */
00070     }
00071
00072     return ret;
00073 }
00074
00075 bool CryptnoxWallet::establishSecureChannel(CW_SecureSession& session) {
00076     bool ret = false;
00077
00078     /* Declare all sensitive stack buffers at function entry so they can be
00079      * wiped on every exit path (H-01, M-02). */
00080     uint8_t cardCertificate[146U]      = { 0U };
00081     uint8_t cardCertificateLength      = 0U;
00082     uint8_t cardEphemeralPubKey[64U]   = { 0U };
00083     uint8_t openSecureChannelSalt[32U] = { 0U };
00084     uint8_t clientPrivateKey[32U]      = { 0U };
00085     uint8_t clientPublicKey[64U]       = { 0U };
00086     CW_Curve sessionCurve              = CW_CURVE_SECP256R1;
00087
00088     if (_secure.selectApdu()) {
00089         /* Fetch the manufacturer certificate BEFORE getCardCertificate().
00090          * The Cryptnox card state machine advances after GET_CARD_CERTIFICATE
00091          * (INS=F8) and will not respond to GET_MANUFACTURER_CERTIFICATE (INS=F7)
00092          * after that point.  Pre-fetching here caches the cert inside
00093          * CW_SecureChannel so that verifyCertificateChain() can use it without
00094          * issuing another APDU. */
00095         if (!_secure.preFetchManufacturerCert()) {
00096 #if CW_DEBUG_LOGGING
00097             _logger.println(F("Failed to pre-fetch manufacturer certificate"));
```

```
00098 #endif
00099         } else {
00100             if (_secure.getCardCertificate(cardCertificate, cardCertificateLength)) {
00101                 uint8_t certResult = _secure.verifyCertificateChain(cardCertificate,
00102                                                                     cardCertificateLength);
00103                 if (certResult != CW_CERT_OK) {
00104 #if CW_DEBUG_LOGGING
00105                     _logger.print(F("Card authenticity check failed (code 0x"));
00106                     _logger.print(certResult, HEX);
00107                     _logger.println(F("). Aborting."));
00108 #endif
00109                 } else {
00110                     if (_secure.extractCardEphemeralKey(cardCertificate, cardEphemeralPubKey)) {
00111                         if (_secure.openSecureChannel(openSecureChannelSalt, clientPublicKey,
00112                                                       clientPrivateKey, sessionCurve)) {
00113                             if (_secure.mutuallyAuthenticate(session, openSecureChannelSalt,
00114                                                              clientPublicKey, clientPrivateKey,
00115                                                              sessionCurve, cardEphemeralPubKey)) {
00116 #if CW_DEBUG_LOGGING
00117                                 _logger.println(F("Secure channel established"));
00118 #endif
00119                                 ret = true;
00120                             } else {
00121 #if CW_DEBUG_LOGGING
00122                                 _logger.println(F("Mutual authentication failed"));
00123 #endif
00124                             }
00125                         } else {
00126 #if CW_DEBUG_LOGGING
00127                             _logger.println(F("Failed to open secure channel"));
00128 #endif
00129                         }
00130                     } else {
00131 #if CW_DEBUG_LOGGING
00132                         _logger.println(F("Failed to extract card ephemeral key"));
00133 #endif
00134                     }
00135                 }
00136             } else {
00137 #if CW_DEBUG_LOGGING
00138                 _logger.println(F("Failed to get card certificate"));
00139 #endif
00140             }
00141         } /* end preFetchManufacturerCert else */
00142     } else {
00143 #if CW_DEBUG_LOGGING
00144         _logger.println(F("Failed to select Cryptnox application"));
00145 #endif
00146     }
00147
00148     /* Wipe all sensitive ephemeral key material on every exit path (H-01, M-02). */
00149     CW_Utils::secure_wipe(clientPrivateKey,      sizeof(clientPrivateKey));
00150     CW_Utils::secure_wipe(openSecureChannelSalt, sizeof(openSecureChannelSalt));
00151     CW_Utils::secure_wipe(clientPublicKey,       sizeof(clientPublicKey));
00152     CW_Utils::secure_wipe(cardEphemeralPubKey,   sizeof(cardEphemeralPubKey));
00153     CW_Utils::secure_wipe(cardCertificate,       sizeof(cardCertificate));
00154
00155     return ret;
00156 }
00157
00158 void CryptnoxWallet::disconnect(CW_SecureSession& session) {
00159     if (isSecureChannelOpen(session)) {
00160         session.clear();
00161     }
00162     _secure.resetReader();
00163 }
00164
00165 bool CryptnoxWallet::getCardInfo(CW_SecureSession& session, CW_CardInfo* info) {
00166     bool ret = false;
00167     if (!isSecureChannelOpen(session)) {
00168 #if CW_DEBUG_LOGGING
00169         _logger.println(F("Error: Secure channel not open. Cannot get card info."));
00170 #endif
00171         return false;
00172     }
00173     uint8_t data[] = { 0x00U };
00174     uint8_t apdu[] = { 0x80U, 0xFAU, 0x00U, 0x00U };
00175
00176     uint8_t  decrypted[255U] = { 0U };
00177     uint16_t decryptedLen    = 0U;
00178
00179     ret = _secure.aesCbcEncrypt(session, apdu, sizeof(apdu),
00180                                 data, sizeof(data),
00181                                 decrypted, &decryptedLen);
00182
00183     if (ret && (info != NULL)) {
00184         /* Response layout (Cryptnox basic_g1 spec):
```

```
00185          *   [byte0] [name_len(1)] [name(name_len)]
00186          *           [email_len(1)] [email(email_len)] [... more fields ...]
00187          * byte0 = unused/flags. */
00188         ret = false;
00189         if (decryptedLen >= 4U) {
00190             uint16_t pos     = 1U;
00191             uint8_t  nameLen = decrypted[pos];
00192             pos += 1U;
00193             if ((nameLen <= CW_CARD_NAME_MAX_LEN) &&
00194                 ((uint16_t)(pos + nameLen + 1U) <= decryptedLen)) {
00195                 (void)CW_Utils::safe_memcpy(reinterpret_cast<uint8_t*>(info->name),
00196                                             sizeof(info->name),
00197                                             decrypted + pos, nameLen);
00198                 info->name[nameLen] = '\0';
00199                 pos += nameLen;
00200
00201                 uint8_t emailLen = decrypted[pos];
00202                 pos += 1U;
00203                 if ((emailLen <= CW_CARD_EMAIL_MAX_LEN) &&
00204                     ((uint16_t)(pos + emailLen) <= decryptedLen)) {
00205                     (void)CW_Utils::safe_memcpy(reinterpret_cast<uint8_t*>(info->email),
00206                                                 sizeof(info->email),
00207                                                 decrypted + pos, emailLen);
00208                     info->email[emailLen] = '\0';
00209                     ret = true;
00210                 }
00211             }
00212         }
00213     }
00214
00215     CW_Utils::secure_wipe(decrypted, sizeof(decrypted));
00216     return ret;
00217 }
00218
00219 bool CryptnoxWallet::verifyPin(CW_SecureSession& session, const uint8_t* pin, uint8_t pinLength) {
00220     bool ret = false;
00221     if (!isSecureChannelOpen(session)) {
00222 #if CW_DEBUG_LOGGING
00223         _logger.println(F("Error: Secure channel not open. Cannot verify PIN."));
00224 #endif
00225     }
00226     else if ((pin == NULL) || (pinLength < CW_MIN_PIN_LENGTH) || (pinLength > CW_MAX_PIN_LENGTH)) {
00227 #if CW_DEBUG_LOGGING
00228         _logger.println(F("Error: Invalid PIN (must be 4-9 digits)."));
00229 #endif
00230     }
00231     else {
00232         uint8_t paddedPin[CW_MAX_PIN_LENGTH] = { 0U };
00233         (void)CW_Utils::safe_memcpy(paddedPin, sizeof(paddedPin), pin, pinLength);
00234         uint8_t apdu[] = { 0x80U, 0x20U, 0x00U, 0x00U };
00235         ret = _secure.aesCbcEncrypt(session, apdu, sizeof(apdu), paddedPin, CW_MAX_PIN_LENGTH);
00236         CW_Utils::secure_wipe(paddedPin, sizeof(paddedPin));
00237     }
00238     return ret;
00239 }
00240
00241 bool CryptnoxWallet::writeUserData(CW_SecureSession& session, uint8_t slot,
00242                                    const uint8_t* data, uint16_t dataLength) {
00243     bool ret = false;
00244
00245     if (!isSecureChannelOpen(session)) {
00246 #if CW_DEBUG_LOGGING
00247         _logger.println(F("Error: Secure channel not open. Cannot write user data."));
00248 #endif
00249     }
00250     else if ((data == NULL) || (dataLength == 0U)) {
00251 #if CW_DEBUG_LOGGING
00252         _logger.println(F("Error: Invalid data for write user data."));
00253 #endif
00254     }
00255     else {
00256         uint16_t offset = 0U;
00257         uint8_t  page   = 0U;
00258         ret = true;
00259
00260         while ((offset < dataLength) && ret) {
00261             uint16_t chunkSize = dataLength - offset;
00262             if (chunkSize > CW_USER_DATA_PAGE_SIZE) {
00263                 chunkSize = CW_USER_DATA_PAGE_SIZE;
00264             }
00265
00266             uint8_t apdu[] = { 0x80U, 0xFCU, slot, page };
00267
00268 #if CW_DEBUG_LOGGING
00269             _logger.print(F("Writing user data page "));
00270             _logger.print(page);
00271             _logger.print(F(" ("));
```

```
00272             _logger.print(chunkSize);
00273             _logger.println(F(" bytes)..."));
00274 #endif
00275 
00276             if (!_secure.aesCbcEncrypt(session, apdu, sizeof(apdu), data + offset, chunkSize)) {
00277 #if CW_DEBUG_LOGGING
00278                 _logger.print(F("Error: Write user data failed on page "));
00279                 _logger.println(page);
00280 #endif
00281                 ret = false;
00282             }
00283             else {
00284                 offset += chunkSize;
00285                 page++;
00286             }
00287         }
00288     }
00289 
00290     return ret;
00291 }
00292 
00293 CW_SignResult CryptnoxWallet::sign(CW_SignRequest& request) {
00294     CW_SignResult result;
00295 
00296     if (validateSignRequest(request, result)) {
00297         uint8_t data[CW_HASH_SIZE + CW_MAX_DERIVE_PATH_LENGTH + CW_MAX_PIN_LENGTH] = { 0U };
00298         uint16_t dataLength = 0U;
00299 
00300         buildSignPayload(request, data, dataLength);
00301 
00302         uint8_t derResponse[255U] = { 0U };
00303         uint16_t derLength = 0U;
00304 
00305         if (sendSignApdu(request, data, dataLength, derResponse, derLength, result)) {
00306             if (extractRawSignature(derResponse, derLength, result)) {
00307                 debugPrintSignature(result.signature);
00308                 result.errorCode = CW_OK;
00309             }
00310         }
00311         CW_Utils::secure_wipe(data, sizeof(data));
00312         CW_Utils::secure_wipe(derResponse, sizeof(derResponse));
00313     }
00314 
00315     return result;
00316 }
00317 
00318 /**********************************************************************
00319  * Static public methods
00320  **********************************************************************/
00321 
00322 bool CryptnoxWallet::parseDerSignature(const uint8_t* der, uint8_t derLength,
00323                                        uint8_t* r, uint8_t& rLength,
00324                                        uint8_t* s, uint8_t& sLength) {
00325     bool ret = false;
00326 
00327     if ((der == NULL) || (derLength < 6U) || (r == NULL) || (s == NULL)) {
00328     }
00329     else if (der[0] != CW_DER_TAG_SEQUENCE) {
00330     }
00331     else {
00332         uint8_t pos = 2U;
00333 
00334         if (der[pos] != CW_DER_TAG_INTEGER) {
00335         }
00336         else {
00337             pos++;
00338             rLength = der[pos];
00339             pos++;
00340             if ((rLength > 33U) || ((pos + rLength) > derLength)) {
00341             }
00342             else {
00343                 (void)CW_Utils::safe_memcpy(r, 33U, der + pos, rLength);
00344                 pos += rLength;
00345 
00346                 if ((pos >= derLength) || (der[pos] != CW_DER_TAG_INTEGER)) {
00347                 }
00348                 else {
00349                     pos++;
00350                     sLength = der[pos];
00351                     pos++;
00352                     if ((sLength > 33U) || ((pos + sLength) > derLength)) {
00353                     }
00354                     else {
00355                         (void)CW_Utils::safe_memcpy(s, 33U, der + pos, sLength);
00356                         ret = true;
00357                     }
00358                 }
```

```
00359             }
00360         }
00361     }
00362
00363     return ret;
00364 }
00365
00366 /**************************************************************************
00367  * Private methods
00368  *************************************************************************/
00369
00370 bool CryptnoxWallet::isSecureChannelOpen(const CW_SecureSession& session) const {
00371     uint8_t acc = 0U;
00372     for (uint8_t i = 0U; i < CW_AESKEY_SIZE; i++) {
00373         acc |= session.aesKey[i];
00374     }
00375     return (acc != 0U);
00376 }
00377
00378 bool CryptnoxWallet::printPN532FirmwareVersion() {
00379     return _secure.printFirmwareVersion();
00380 }
00381
00382 bool CryptnoxWallet::validateSignRequest(const CW_SignRequest& request, CW_SignResult& result) {
00383     bool ret = false;
00384
00385     if (!isSecureChannelOpen(request.session)) {
00386 #if CW_DEBUG_LOGGING
00387         _logger.println(F("Error: Secure channel not open. Cannot sign."));
00388 #endif
00389         result.errorCode = CW_INVALID_SESSION;
00390     }
00391     else if ((request.hash == NULL) || (request.hashLength == 0U)) {
00392 #if CW_DEBUG_LOGGING
00393         _logger.println(F("Error: Invalid parameters for sign."));
00394 #endif
00395         result.errorCode = CW_SIGN_KEY_TOO_SHORT;
00396     }
00397     else if (request.hashLength > CW_HASH_SIZE) {
00398 #if CW_DEBUG_LOGGING
00399         _logger.println(F("Error: Hash too large."));
00400 #endif
00401         result.errorCode = CW_SIGN_KEY_TOO_SHORT;
00402     }
00403     else if ((request.pinLessMode) && (request.keyType != CW_SIGN_PINLESS_K1)) {
00404 #if CW_DEBUG_LOGGING
00405         _logger.println(F("Error: PIN-less mode requires CW_SIGN_PINLESS_K1 key type."));
00406 #endif
00407         result.errorCode = CW_SIGN_KEY_TOO_SHORT_WITH_PINLESS_MODE;
00408     }
00409     else {
00410         ret = true;
00411
00412         if (!request.pinLessMode) {
00413             uint8_t pinLength = 0U;
00414             for (uint8_t i = 0U; i < CW_MAX_PIN_LENGTH; i++) {
00415                 if (request.pin[i] == 0U) { break; }
00416                 pinLength++;
00417             }
00418             if ((pinLength > 0U) && (pinLength < CW_MIN_PIN_LENGTH)) {
00419 #if CW_DEBUG_LOGGING
00420                 _logger.println(F("Error: PIN too short (must be 4-9 digits)."));
00421 #endif
00422                 result.errorCode = CW_SIGN_PIN_INCORRECT;
00423                 ret = false;
00424             }
00425         }
00426     }
00427
00428     return ret;
00429 }
00430
00431 void CryptnoxWallet::buildSignPayload(const CW_SignRequest& request,
00432                                       uint8_t* data, uint16_t& dataLength) {
00433     const size_t kDataBufSize = static_cast<size_t>(CW_HASH_SIZE) +
    static_cast<size_t>(CW_MAX_DERIVE_PATH_LENGTH) + static_cast<size_t>(CW_MAX_PIN_LENGTH);
00434     dataLength = request.hashLength;
00435     (void)CW_Utils::safe_memcpy(data, kDataBufSize, request.hash, request.hashLength);
00436
00437     if ((request.keyType == CW_SIGN_DERIVE_K1 || request.keyType == CW_SIGN_DERIVE_R1) &&
00438         (request.derivePath != NULL) && (request.derivePathLength > 0U)) {
00439         (void)CW_Utils::safe_memcpy(data + dataLength, kDataBufSize - static_cast<size_t>(dataLength),
    request.derivePath, request.derivePathLength);
00440         dataLength += request.derivePathLength;
00441     }
00442
00443     if (!request.pinLessMode) {
```

```
00444         uint8_t pinLength = 0U;
00445         for (uint8_t i = 0U; i < CW_MAX_PIN_LENGTH; i++) {
00446             if (request.pin[i] == 0U) { break; }
00447             pinLength++;
00448         }
00449         if (pinLength > 0U) {
00450             (void)CW_Utils::safe_memcpy(data + dataLength, kDataBufSize -
      static_cast<size_t>(dataLength), request.pin, CW_MAX_PIN_LENGTH);
00451             dataLength += CW_MAX_PIN_LENGTH;
00452         }
00453     }
00454 }
00455
00456 bool CryptnoxWallet::sendSignApdu(CW_SignRequest& request, const uint8_t* data,
00457                                   uint16_t dataLength, uint8_t* derResponse,
00458                                   uint16_t& derLength, CW_SignResult& result) {
00459     bool ret = false;
00460     uint8_t apdu[] = { 0x80U, 0xC0U, request.keyType, request.signatureType };
00461
00462 #if CW_DEBUG_LOGGING
00463     _logger.println(F("Sending SIGN APDU..."));
00464 #endif
00465
00466     if (_secure.aesCbcEncrypt(request.session, apdu, sizeof(apdu), data, dataLength,
00467                               derResponse, &derLength)) {
00468         ret = true;
00469     }
00470     else {
00471 #if CW_DEBUG_LOGGING
00472         _logger.println(F("Sign APDU failed."));
00473 #endif
00474         result.errorCode = CW_SIGN_NO_KEY_LOADED;
00475     }
00476
00477     return ret;
00478 }
00479
00480 bool CryptnoxWallet::extractRawSignature(const uint8_t* derResponse, uint16_t derLength,
00481                                          CW_SignResult& result) {
00482     bool ret = false;
00483
00484     if ((derLength < 2U) || (derResponse[0] != CW_DER_TAG_SEQUENCE)) {
00485 #if CW_DEBUG_LOGGING
00486         _logger.println(F("Error: Invalid signature data (missing DER SEQUENCE tag)."));
00487 #endif
00488         result.errorCode = CW_NOK;
00489     }
00490     else {
00491         uint8_t derContentLength = derResponse[1];
00492         uint8_t derTotalLength   = 2U + derContentLength;
00493
00494         if (derTotalLength > derLength) {
00495 #if CW_DEBUG_LOGGING
00496             _logger.println(F("Error: DER signature length exceeds response."));
00497 #endif
00498             result.errorCode = CW_NOK;
00499         }
00500         else {
00501             uint8_t r[33U] = { 0U };
00502             uint8_t s[33U] = { 0U };
00503             uint8_t rLen = 0U;
00504             uint8_t sLen = 0U;
00505
00506             if (!parseDerSignature(derResponse, derTotalLength, r, rLen, s, sLen)) {
00507 #if CW_DEBUG_LOGGING
00508                 _logger.println(F("Error: Failed to parse DER signature."));
00509 #endif
00510                 result.errorCode = CW_NOK;
00511             }
00512             else {
00513                 memset(result.signature, 0U, CW_RAW_SIGNATURE_SIZE);
00514
00515                 if (rLen > 0U) {
00516                     uint8_t rSrc = 0U;
00517                     uint8_t rDstLen = 32U;
00518                     if ((rLen == 33U) && (r[0] == 0x00U)) { rSrc = 1U; rLen = 32U; }
00519                     if (rLen <= rDstLen) {
00520                         (void)CW_Utils::safe_memcpy(result.signature + (rDstLen - rLen),
      static_cast<size_t>(rDstLen + rLen), r + rSrc, rLen);
00521                     }
00522                 }
00523
00524                 if (sLen > 0U) {
00525                     uint8_t sSrc = 0U;
00526                     uint8_t sDstLen = 32U;
00527                     if ((sLen == 33U) && (s[0] == 0x00U)) { sSrc = 1U; sLen = 32U; }
00528                     if (sLen <= sDstLen) {
```

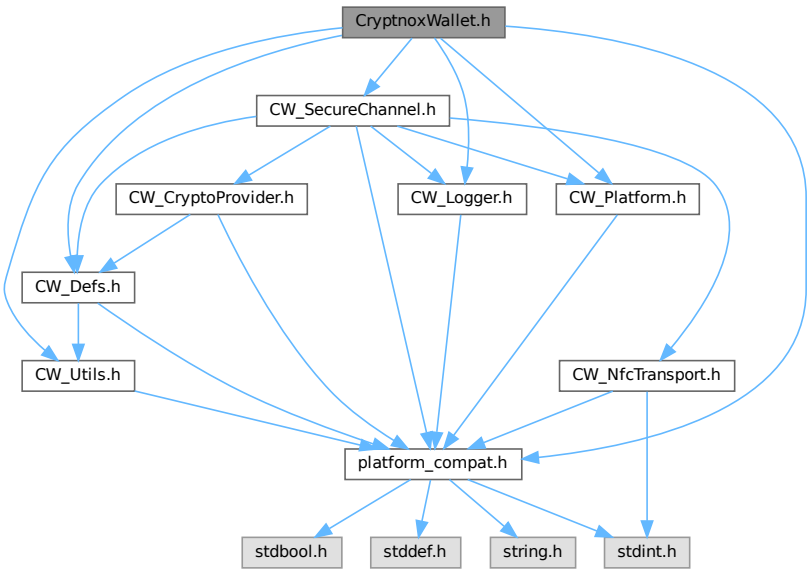
```
00529                         (void)CW_Utils::safe_memcpy(result.signature + 32U + (sDstLen - sLen),
      static_cast<size_t>(sLen), s + sSrc, sLen);
00530                     }
00531                 }
00532
00533                 ret = true;
00534             }
00535             CW_Utils::secure_wipe(r, sizeof(r));
00536             CW_Utils::secure_wipe(s, sizeof(s));
00537         }
00538     }
00539
00540     return ret;
00541 }
00542
00543 void CryptnoxWallet::debugPrintSignature(const uint8_t* signature) {
00544 #if CW_DEBUG_LOGGING
00545     _logger.print(F("Signature ("));
00546     _logger.print((uint8_t)CW_RAW_SIGNATURE_SIZE);
00547     _logger.println(F(" bytes):"));
00548     for (uint8_t i = 0U; i < CW_RAW_SIGNATURE_SIZE; i++) {
00549         _logger.print(F("0x"));
00550         if (signature[i] < 0x10U) { _logger.print(F("0")); }
00551         _logger.print(signature[i], HEX);
00552         _logger.print(F(" "));
00553         if (((i + 1U) % 16U == 0U) && ((i + 1U) != CW_RAW_SIGNATURE_SIZE)) { _logger.println(); }
00554     }
00555     _logger.println();
00556 #else
00557     (void)signature;
00558 #endif
00559 }
```

4.3 CryptnoxWallet.h File Reference

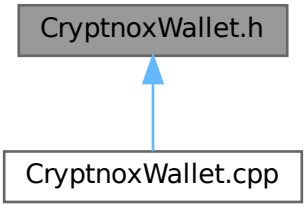
High-level API for interacting with a Cryptnox Hardware Wallet over NFC.

```
#include "platform_compat.h"
#include "CW_Defs.h"
#include "CW_Logger.h"
#include "CW_Platform.h"
#include "CW_SecureChannel.h"
#include "CW_Utils.h"
```

Include dependency graph for CryptnoxWallet.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct CW_CardInfo

  Subset of the Cryptnox card info returned by APDU 0x80FA0000.
- struct CW_SignRequest

  Request parameters for CryptnoxWallet::sign.
- struct CW_SignResult

  Result of CryptnoxWallet::sign.
- class CryptnoxWallet

  High-level interface for interacting with a Cryptnox Hardware Wallet over NFC.

Macros

- #define CW_CARD_NAME_MAX_LEN (20U)

  Max name length stored on a Cryptnox card (per card spec).
- #define CW_CARD_EMAIL_MAX_LEN (60U)

  Max email length stored on a Cryptnox card (per card spec).

4.3.1 Detailed Description

High-level API for interacting with a Cryptnox Hardware Wallet over NFC.
Declares CryptnoxWallet, the main entry point for application code. The class wires together the four abstract adapters supplied by the host integration (NFC transport, crypto provider, logger, platform) and exposes the wallet operations: card connection, secure channel establishment, card info retrieval, PIN verification, transaction signing, and user-data writing.

See also

CW_NfcTransport

CW_CryptoProvider

CW_Logger

CW_Platform

Definition in file CryptnoxWallet.h.

4.3.2 Macro Definition Documentation

4.3.2.1 CW_CARD_EMAIL_MAX_LEN

```
#define CW_CARD_EMAIL_MAX_LEN (60U)
```

Max email length stored on a Cryptnox card (per card spec).
Definition at line 45 of file CryptnoxWallet.h.
Referenced by CryptnoxWallet::getCardInfo().

4.3.2.2 CW_CARD_NAME_MAX_LEN

```
#define CW_CARD_NAME_MAX_LEN (20U)
```

Max name length stored on a Cryptnox card (per card spec).
Definition at line 42 of file CryptnoxWallet.h.
Referenced by CryptnoxWallet::getCardInfo().

4.4 CryptnoxWallet.h

Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00022
00023 #ifndef CRYPTNOXWALLET_H
00024 #define CRYPTNOXWALLET_H
00025
00026 /**********************************************************************
00027  * 1. Included files
00028  **********************************************************************/
00029
00030 #include "platform_compat.h"
00031 #include "CW_Defs.h"
00032 #include "CW_Logger.h"
00033 #include "CW_Platform.h"
00034 #include "CW_SecureChannel.h"
00035 #include "CW_Utils.h"
00036
00037 /**********************************************************************
00038  * 2. Typedefs / structs (sign API)
00039  **********************************************************************/
00040
00042 #define CW_CARD_NAME_MAX_LEN  (20U)
00043
00045 #define CW_CARD_EMAIL_MAX_LEN (60U)
00046
00055 struct CW_CardInfo {
00056     char name[CW_CARD_NAME_MAX_LEN + 1U];
00057     char email[CW_CARD_EMAIL_MAX_LEN + 1U];
00058
00059     CW_CardInfo() {
00060         name[0]  = '\0';
00061         email[0] = '\0';
00062     }
00063 };
00064
00074 struct CW_SignRequest {
00075     CW_SecureSession& session;
00076     uint8_t keyType;
00077     uint8_t signatureType;
00078     uint8_t pin[CW_MAX_PIN_LENGTH];
00079     bool pinLessMode;
00080     const uint8_t* hash;
00081     uint8_t hashLength;
00082     const uint8_t* derivePath;
00083     uint8_t derivePathLength;
00084
00092     explicit CW_SignRequest(CW_SecureSession& sess,
00093                             uint8_t kType   = CW_SIGN_CURR_K1,
00094                             uint8_t sigType = CW_SIGN_SIG_ECDSA_LOW_S,
00095                             bool pinless    = CW_SIGN_WITH_PIN)
00096         : session(sess), keyType(kType), signatureType(sigType),
00097           pinLessMode(pinless), hash(NULL), hashLength(0U),
00098           derivePath(NULL), derivePathLength(0U) {
00099         memset(pin, 0U, sizeof(pin));
00100     }
```

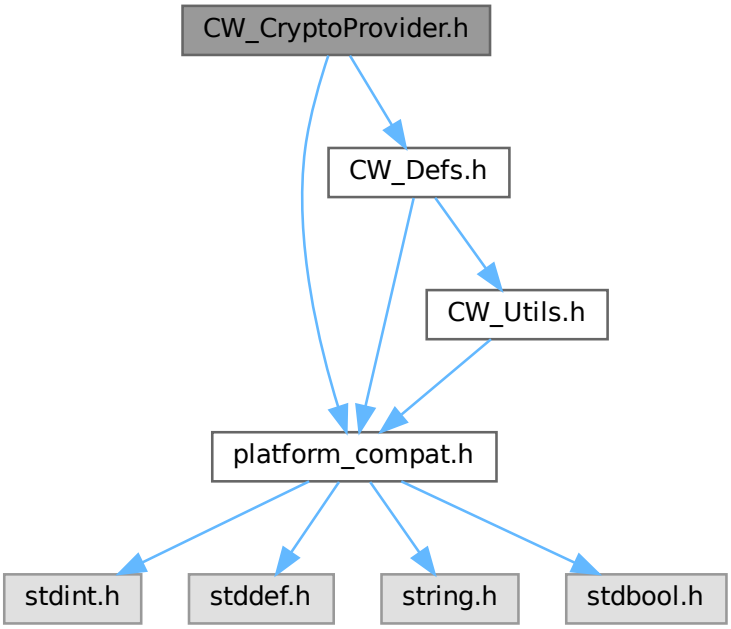
```
00101
00103     ~CW_SignRequest() {
00104         CW_Utils::secure_wipe(pin, sizeof(pin));
00105     }
00106 };
00107
00116 struct CW_SignResult {
00117     uint8_t signature[CW_RAW_SIGNATURE_SIZE];
00118     uint8_t errorCode;
00119
00121     CW_SignResult() : errorCode(CW_NOK) {
00122         memset(signature, 0U, sizeof(signature));
00123     }
00124 };
00125
00126 /**************************************************************
00127  * 3. CryptnoxWallet class
00128  **************************************************************/
00129
00160 class CryptnoxWallet {
00161 public:
00170     CryptnoxWallet(CW_NfcTransport& driver, CW_Logger& logger,
00171                    CW_CryptoProvider& crypto, CW_Platform& platform);
00172
00173     CryptnoxWallet(const CryptnoxWallet&) = delete;
00174     CryptnoxWallet& operator=(const CryptnoxWallet&) = delete;
00175
00180     bool begin();
00181
00200     bool connect(CW_SecureSession& session);
00201
00217     bool establishSecureChannel(CW_SecureSession& session);
00218
00232     void disconnect(CW_SecureSession& session);
00233
00245     bool getCardInfo(CW_SecureSession& session, CW_CardInfo* info = NULL);
00246
00267     bool verifyPin(CW_SecureSession& session, const uint8_t* pin, uint8_t pinLength);
00268
00298     CW_SignResult sign(CW_SignRequest& request);
00299
00309     bool writeUserData(CW_SecureSession& session, uint8_t slot,
00310                        const uint8_t* data, uint16_t dataLength);
00311
00323     static bool parseDerSignature(const uint8_t* der, uint8_t derLength,
00324                                   uint8_t* r, uint8_t& rLength,
00325                                   uint8_t* s, uint8_t& sLength);
00326
00327 private:
00328     CW_Logger&       _logger;
00329     CW_Platform&     _platform;
00330     CW_SecureChannel _secure;
00331
00332     bool isSecureChannelOpen(const CW_SecureSession& session) const;
00333     bool printPN532FirmwareVersion();
00334
00335     /* Sign helper methods */
00336     bool validateSignRequest(const CW_SignRequest& request, CW_SignResult& result);
00337     void buildSignPayload(const CW_SignRequest& request, uint8_t* data, uint16_t& dataLength);
00338     bool sendSignApdu(CW_SignRequest& request, const uint8_t* data, uint16_t dataLength,
00339                       uint8_t* derResponse, uint16_t& derLength, CW_SignResult& result);
00340     bool extractRawSignature(const uint8_t* derResponse, uint16_t derLength, CW_SignResult& result);
00341     void debugPrintSignature(const uint8_t* signature);
00342 };
00343
00344 #endif // CRYPTNOXWALLET_H
```

4.5 CW_CryptoProvider.h File Reference

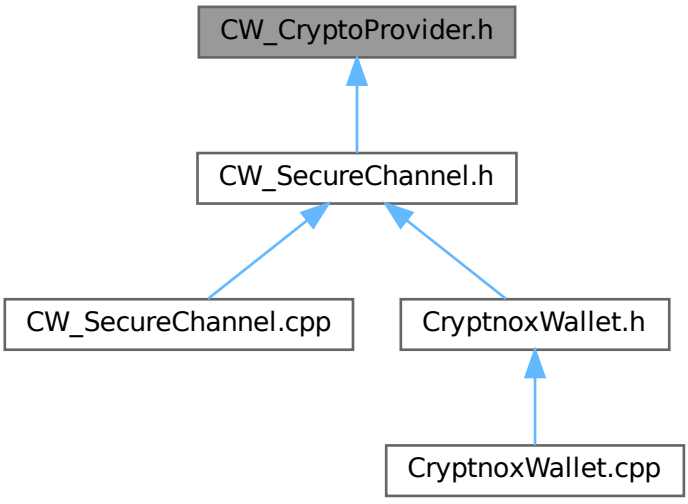
Abstract cryptographic primitives interface.

```
#include "platform_compat.h"
#include "CW_Defs.h"
```

Include dependency graph for CW_CryptoProvider.h:



This graph shows which files directly or indirectly include this file:



Classes

- class CW_CryptoProvider

Abstract interface for cryptographic operations used by CW_SecureChannel.

4.5.1 Detailed Description

Abstract cryptographic primitives interface.
Declares CW_CryptoProvider, the contract that any concrete cryptographic backend (mbedTLS, BearSSL, OpenSSL, ESP32 hardware crypto, micro-ecc + AESLib + SHA512, ...) must implement so the secure channel remains decoupled from any specific crypto library.
Provides: SHA-256/512, AES-CBC encrypt/decrypt, ECDH shared secret, EC key pair generation, and cryptographically secure RNG.
One of the three adapter interfaces a host integration must provide.
Definition in file CW_CryptoProvider.h.

4.6 CW_CryptoProvider.h

Go to the documentation of this file.

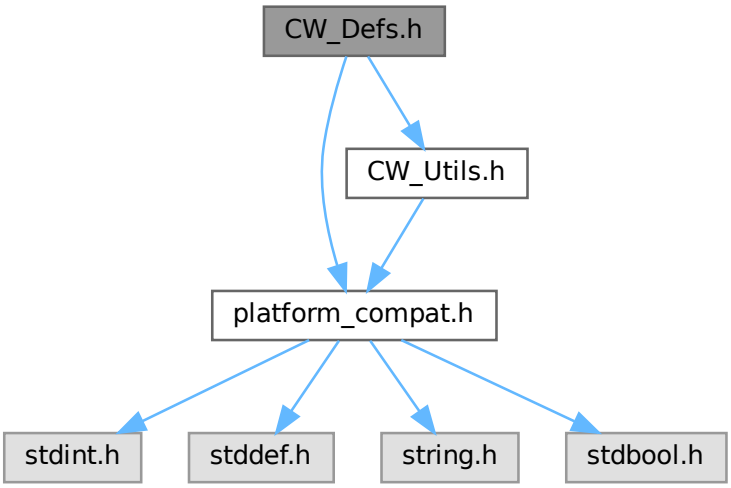
```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00020
00021 #ifndef CW_CRYPTOPROVIDER_H
00022 #define CW_CRYPTOPROVIDER_H
00023
00024 /**************************************************************
00025  * 1. Included files
00026  **************************************************************/
00027
00028 #include "platform_compat.h"
00029 #include "CW_Defs.h"
00030
00031 /**************************************************************
00032  * 2. Class declaration
00033  **************************************************************/
00034
00045 class CW_CryptoProvider {
00046 public:
00055     virtual bool sha256(const uint8_t* data, size_t len, uint8_t* out) = 0;
00056
00065     virtual bool sha512(const uint8_t* data, size_t len, uint8_t* out) = 0;
00066
00080     virtual uint16_t aesCbcEncrypt(const uint8_t* in, uint16_t len, uint8_t* out,
00081                                    const uint8_t* key, uint8_t keyLen,
00082                                    uint8_t* iv, bool bitPadding) = 0;
00083
00096     virtual uint16_t aesCbcDecrypt(uint8_t* in, uint16_t len, uint8_t* out,
00097                                    const uint8_t* key, uint8_t keyLen,
00098                                    uint8_t* iv, bool bitPadding) = 0;
00099
00109     virtual bool ecdh(const uint8_t* pubKey, const uint8_t* privKey,
00110                       uint8_t* secret, CW_Curve curve) = 0;
00111
00120     virtual bool makeKey(uint8_t* pubKey, uint8_t* privKey,
00121                          CW_Curve curve) = 0;
00122
00130     virtual bool random(uint8_t* dest, unsigned size) = 0;
00131
00142     virtual bool ecdsaVerify(const uint8_t* pubKey64,
00143                              const uint8_t* hash, size_t hashLen,
00144                              const uint8_t* sig, CW_Curve curve) = 0;
00145
00146     virtual ~CW_CryptoProvider() {}
00147 };
00148
00149 #endif // CW_CRYPTOPROVIDER_H
```

4.7 CW_Defs.h File Reference

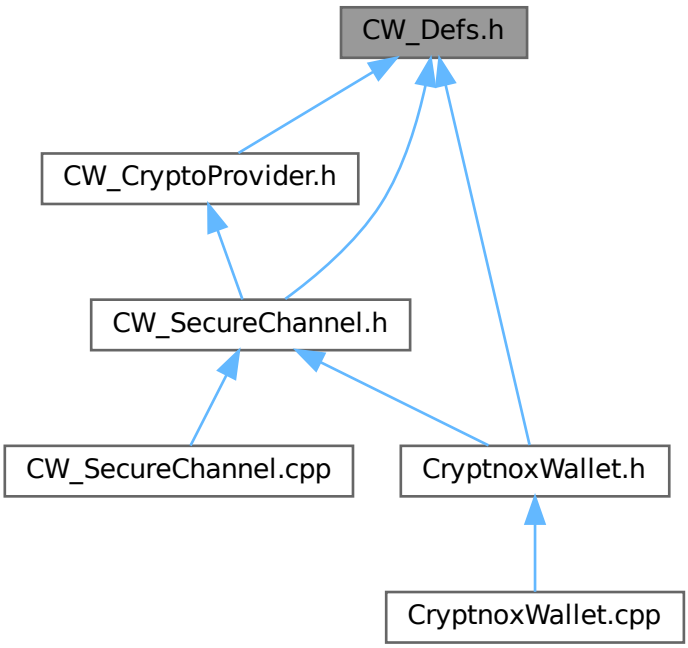
Shared constants, error codes, and session state for the SDK.

```
#include "platform_compat.h"
#include "CW_Utils.h"
```

Include dependency graph for CW_Defs.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct CW_SecureSession

Holds cryptographic session state for reentrant secure channel operations.

Macros

- #define CW_AESKEY_SIZE (32U)
- #define CW_MACKEY_SIZE (32U)
- #define CW_IV_SIZE (16U)
- #define CW_OK (0x00U)
- #define CW_NOK (0x01U)
- #define CW_INVALID_SESSION (0x02U)
- #define CW_SIGN_CURR_K1 (0x00U)
- #define CW_SIGN_CURR_R1 (0x10U)
- #define CW_SIGN_DERIVE_K1 (0x01U)
- #define CW_SIGN_DERIVE_R1 (0x11U)
- #define CW_SIGN_PINLESS_K1 (0x03U)
- #define CW_SIGN_WITH_PIN (false)
- #define CW_SIGN_PINLESS (true)
- #define CW_SIGN_SIG_ECDSA_LOW_S (0x00U)
- #define CW_SIGN_SIG_ECDSA_EOSIO (0x01U)
- #define CW_SIGN_SIG_SCHNORR_BIP340 (0x02U)
- #define CW_SIGN_KEY_TOO_SHORT (0x80U)
- #define CW_SIGN_NO_KEY_LOADED (0x81U)
- #define CW_SIGN_PIN_INCORRECT (0x82U)
- #define CW_SIGN_KEY_TOO_SHORT_WITH_PINLESS_MODE (0x83U)
- #define CW_RAW_SIGNATURE_SIZE (64U)
- #define CW_HASH_SIZE (32U)
- #define CW_MAX_DERIVE_PATH_LENGTH (20U)
- #define CW_MIN_PIN_LENGTH (4U)
- #define CW_MAX_PIN_LENGTH (9U)
- #define CW_USER_DATA_PAGE_SIZE (208U)
- #define CW_CONNECT_MAX_ATTEMPTS (5U)
- #define CW_SIG_R_OFFSET (0U)
- #define CW_SIG_S_OFFSET (32U)
- #define CW_DER_TAG_SEQUENCE (0x30U)
- #define CW_DER_TAG_INTEGER (0x02U)
- #define CW_CERT_NONCE_SIZE (8U)
- #define CW_CERT_OK (0x00U)
- #define CW_CERT_FORMAT_ERROR (0x10U)
- #define CW_CERT_NONCE_MISMATCH (0x11U)
- #define CW_CERT_CARD_SIG_INVALID (0x12U)
- #define CW_CERT_MANUF_SIG_INVALID (0x13U)
- #define CW_CERT_KEY_NOT_FOUND (0x14U)
- #define CW_MANUF_CERT_MAX_BYTES (420U)
- #define CW_VERIFY_CERT 1
- #define CW_DEBUG_LOGGING 0

Enumerations

- enum CW_Curve { CW_CURVE_SECP256R1 = 0 , CW_CURVE_SECP256K1 = 1 }

  Portable curve identifier used throughout the SDK.

4.7.1 Detailed Description

Shared constants, error codes, and session state for the SDK.
Defines:

- AES session key / IV sizes
- Generic and SIGN-specific error codes (`CW_OK`, `CW_NOK`, `CW_SIGN_*`)
- SIGN APDU parameter values (key types, signature types, PIN/PIN-less)
- Buffer and protocol size limits
- Certificate verification result codes (`CW_CERT_*`)
- CW_Curve enum (portable curve identifier)
- CW_SecureSession (encryption + MAC keys + rolling IV)
- Compile-time security gates: `CW_VERIFY_CERT`, `CW_DEBUG_LOGGING`

Definition in file CW_Defs.h.

4.7.2 Macro Definition Documentation

4.7.2.1 CW_AESKEY_SIZE

```
#define CW_AESKEY_SIZE (32U)
```

AES-256 session encryption key size in bytes
Definition at line 75 of file CW_Defs.h.
Referenced by CryptnoxWallet::isSecureChannelOpen(), and CW_SecureChannel::mutuallyAuthenticate().

4.7.2.2 CW_CERT_CARD_SIG_INVALID

```
#define CW_CERT_CARD_SIG_INVALID (0x12U)
```

Card cert ECDSA sig failed
Definition at line 130 of file CW_Defs.h.
Referenced by CW_SecureChannel::verifyCertificateChain().

4.7.2.3 CW_CERT_FORMAT_ERROR

```
#define CW_CERT_FORMAT_ERROR (0x10U)
```

Malformed certificate data
Definition at line 128 of file CW_Defs.h.
Referenced by CW_SecureChannel::verifyCertificateChain().

4.7.2.4 CW_CERT_KEY_NOT_FOUND

```
#define CW_CERT_KEY_NOT_FOUND (0x14U)
```

Device public key OID not found
Definition at line 132 of file CW_Defs.h.
Referenced by CW_SecureChannel::verifyCertificateChain().

4.7.2.5 CW_CERT_MANUF_SIG_INVALID

```
#define CW_CERT_MANUF_SIG_INVALID (0x13U)
```

Manufacturer cert ECDSA sig failed
Definition at line 131 of file CW_Defs.h.
Referenced by CW_SecureChannel::verifyCertificateChain().

4.7.2.6 CW_CERT_NONCE_MISMATCH

```
#define CW_CERT_NONCE_MISMATCH (0x11U)
```

Challenge nonce not echoed
Definition at line 129 of file CW_Defs.h.
Referenced by CW_SecureChannel::verifyCertificateChain().

4.7.2.7 CW_CERT_NONCE_SIZE

```
#define CW_CERT_NONCE_SIZE (8U)
```

Challenge nonce length in bytes
Definition at line 124 of file CW_Defs.h.
Referenced by CW_SecureChannel::verifyCertificateChain().

4.7.2.8 CW_CERT_OK

```
#define CW_CERT_OK (0x00U)
```

Certificate chain verified
Definition at line 127 of file CW_Defs.h.
Referenced by CryptnoxWallet::establishSecureChannel(), and CW_SecureChannel::verifyCertificateChain().

4.7.2.9 CW_CONNECT_MAX_ATTEMPTS

```
#define CW_CONNECT_MAX_ATTEMPTS (5U)
```

Max NFC connection retry attempts
Definition at line 113 of file CW_Defs.h.
Referenced by CryptnoxWallet::connect().

4.7.2.10 CW_DEBUG_LOGGING

```
#define CW_DEBUG_LOGGING 0
```

Set to 1 to enable library-internal debug logging via CW_Logger.
Off by default. Enabling it kills flash optimisation — measured on Arduino UNO R4 (Renesas RA4↩ M1): +149 KB flash, +6 KB SRAM (31 % → 88 % of a 256 KB image on the Sign example). Same order of magnitude on every constrained MCU.
Also leaks session state over UART (SEC-012). Bring-up only, never in release builds.
Definition at line 215 of file CW_Defs.h.

4.7.2.11 CW_DER_TAG_INTEGER

```
#define CW_DER_TAG_INTEGER (0x02U)
```

Definition at line 121 of file CW_Defs.h.
Referenced by CryptnoxWallet::parseDerSignature().

4.7.2.12 CW_DER_TAG_SEQUENCE

```
#define CW_DER_TAG_SEQUENCE (0x30U)
```

Definition at line 120 of file CW_Defs.h.
Referenced by CryptnoxWallet::extractRawSignature(), and CryptnoxWallet::parseDerSignature().

4.7.2.13 CW_HASH_SIZE

```
#define CW_HASH_SIZE (32U)
```

Standard hash size
Definition at line 108 of file CW_Defs.h.
Referenced by CryptnoxWallet::buildSignPayload(), CryptnoxWallet::sign(), and CryptnoxWallet::validateSignRequest().

4.7.2.14 CW_INVALID_SESSION

```
#define CW_INVALID_SESSION (0x02U)
```

Invalid session
Definition at line 82 of file CW_Defs.h.
Referenced by CryptnoxWallet::validateSignRequest().

4.7.2.15 CW_IV_SIZE

```
#define CW_IV_SIZE (16U)
```

AES-CBC IV size in bytes
Definition at line 77 of file CW_Defs.h.
Referenced by CW_SecureChannel::aesCbcEncrypt(), and CW_SecureChannel::mutuallyAuthenticate().

4.7.2.16 CW_MACKEY_SIZE

```
#define CW_MACKEY_SIZE (32U)
```

AES-256 session MAC key size in bytes
Definition at line 76 of file CW_Defs.h.
Referenced by CW_SecureChannel::mutuallyAuthenticate().

4.7.2.17 CW_MANUF_CERT_MAX_BYTES

```
#define CW_MANUF_CERT_MAX_BYTES (420U)
```

Definition at line 136 of file CW_Defs.h.
Referenced by CW_SecureChannel::getManufacturerCertificate().

4.7.2.18 CW_MAX_DERIVE_PATH_LENGTH

```
#define CW_MAX_DERIVE_PATH_LENGTH (20U)
```

Max BIP32 path bytes
Definition at line 109 of file CW_Defs.h.
Referenced by CryptnoxWallet::buildSignPayload(), and CryptnoxWallet::sign().

4.7.2.19 CW_MAX_PIN_LENGTH

```
#define CW_MAX_PIN_LENGTH (9U)
```

Maximum PIN length
Definition at line 111 of file CW_Defs.h.
Referenced by CryptnoxWallet::buildSignPayload(), CryptnoxWallet::sign(), CryptnoxWallet::validateSignRequest(), and CryptnoxWallet::verifyPin().

4.7.2.20 CW_MIN_PIN_LENGTH

```
#define CW_MIN_PIN_LENGTH (4U)
```

Minimum PIN length
Definition at line 110 of file CW_Defs.h.
Referenced by CryptnoxWallet::validateSignRequest(), and CryptnoxWallet::verifyPin().

4.7.2.21 CW_NOK

```
#define CW_NOK (0x01U)
```

NOK
Definition at line 81 of file CW_Defs.h.
Referenced by CW_SignResult::CW_SignResult(), and CryptnoxWallet::extractRawSignature().

4.7.2.22 CW_OK

```
#define CW_OK (0x00U)
```
OK
Definition at line 80 of file CW_Defs.h.
Referenced by CryptnoxWallet::sign().

4.7.2.23 CW_RAW_SIGNATURE_SIZE

```
#define CW_RAW_SIGNATURE_SIZE (64U)
```
Raw signature (r[32] + s[32])
Definition at line 107 of file CW_Defs.h.
Referenced by CryptnoxWallet::debugPrintSignature(), and CryptnoxWallet::extractRawSignature().

4.7.2.24 CW_SIG_R_OFFSET

```
#define CW_SIG_R_OFFSET (0U)
```
Byte offset of the r component
Definition at line 116 of file CW_Defs.h.

4.7.2.25 CW_SIG_S_OFFSET

```
#define CW_SIG_S_OFFSET (32U)
```
Byte offset of the s component
Definition at line 117 of file CW_Defs.h.

4.7.2.26 CW_SIGN_CURR_K1

```
#define CW_SIGN_CURR_K1 (0x00U)
```
Current key (k1)
Definition at line 85 of file CW_Defs.h.
Referenced by CW_SignRequest::CW_SignRequest().

4.7.2.27 CW_SIGN_CURR_R1

```
#define CW_SIGN_CURR_R1 (0x10U)
```
Current key (r1)
Definition at line 86 of file CW_Defs.h.

4.7.2.28 CW_SIGN_DERIVE_K1

```
#define CW_SIGN_DERIVE_K1 (0x01U)
```
Derive with k1 curve
Definition at line 87 of file CW_Defs.h.
Referenced by CryptnoxWallet::buildSignPayload().

4.7.2.29 CW_SIGN_DERIVE_R1

```
#define CW_SIGN_DERIVE_R1 (0x11U)
```
Derive with r1 curve
Definition at line 88 of file CW_Defs.h.
Referenced by CryptnoxWallet::buildSignPayload().

4.7.2.30 CW_SIGN_KEY_TOO_SHORT

```
#define CW_SIGN_KEY_TOO_SHORT (0x80U)
```
Definition at line 101 of file CW_Defs.h.
Referenced by CryptnoxWallet::validateSignRequest().

4.7.2.31 CW_SIGN_KEY_TOO_SHORT_WITH_PINLESS_MODE

```
#define CW_SIGN_KEY_TOO_SHORT_WITH_PINLESS_MODE (0x83U)
```

Definition at line 104 of file CW_Defs.h.
Referenced by CryptnoxWallet::validateSignRequest().

4.7.2.32 CW_SIGN_NO_KEY_LOADED

```
#define CW_SIGN_NO_KEY_LOADED (0x81U)
```

Definition at line 102 of file CW_Defs.h.
Referenced by CryptnoxWallet::sendSignApdu().

4.7.2.33 CW_SIGN_PIN_INCORRECT

```
#define CW_SIGN_PIN_INCORRECT (0x82U)
```

Definition at line 103 of file CW_Defs.h.
Referenced by CryptnoxWallet::validateSignRequest().

4.7.2.34 CW_SIGN_PINLESS

```
#define CW_SIGN_PINLESS (true)
```

PIN-less path
Definition at line 93 of file CW_Defs.h.

4.7.2.35 CW_SIGN_PINLESS_K1

```
#define CW_SIGN_PINLESS_K1 (0x03U)
```

PIN-less path (k1 only)
Definition at line 89 of file CW_Defs.h.
Referenced by CryptnoxWallet::validateSignRequest().

4.7.2.36 CW_SIGN_SIG_ECDSA_EOSIO

```
#define CW_SIGN_SIG_ECDSA_EOSIO (0x01U)
```

ECDSA EOSIO format
Definition at line 97 of file CW_Defs.h.

4.7.2.37 CW_SIGN_SIG_ECDSA_LOW_S

```
#define CW_SIGN_SIG_ECDSA_LOW_S (0x00U)
```

ECDSA with canonical low S
Definition at line 96 of file CW_Defs.h.
Referenced by CW_SignRequest::CW_SignRequest().

4.7.2.38 CW_SIGN_SIG_SCHNORR_BIP340

```
#define CW_SIGN_SIG_SCHNORR_BIP340 (0x02U)
```

Schnorr BIP340
Definition at line 98 of file CW_Defs.h.

4.7.2.39 CW_SIGN_WITH_PIN

```
#define CW_SIGN_WITH_PIN (false)
```

PIN path
Definition at line 92 of file CW_Defs.h.
Referenced by CW_SignRequest::CW_SignRequest().

4.7.2.40 CW_USER_DATA_PAGE_SIZE

```
#define CW_USER_DATA_PAGE_SIZE (208U)
```

Max plaintext bytes per write user data page
Definition at line 112 of file CW_Defs.h.
Referenced by CryptnoxWallet::writeUserData().

4.7.2.41 CW_VERIFY_CERT

```
#define CW_VERIFY_CERT 1
```

Certificate chain verification is always enabled (SEC-004 / H-07). Building with -DCW_VERIFY_↩ CERT=0 is a hard error — it disables the card authenticity gate and allows any forged key to be accepted.
Definition at line 196 of file CW_Defs.h.

4.8 CW_Defs.h

Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00020
00021 #ifndef CW_DEFS_H
00022 #define CW_DEFS_H
00023
00024 /**************************************************************************
00025  * 0. Doxygen module groups (used by @ingroup throughout the SDK)
00026  *************************************************************************/
00027
00035
00044
00053
00062
00063 /**************************************************************************
00064  * 1. Included files
00065  *************************************************************************/
00066
00067 #include "platform_compat.h"
00068 #include "CW_Utils.h"
00069
00070 /**************************************************************************
00071  * 2. Constants / define declarations
00072  *************************************************************************/
00073
00074 /* Session key sizes */
00075 #define CW_AESKEY_SIZE    (32U)
00076 #define CW_MACKEY_SIZE    (32U)
00077 #define CW_IV_SIZE        (16U)
00078
00079 /* Generic error codes */
00080 #define CW_OK                          (0x00U)
00081 #define CW_NOK                         (0x01U)
00082 #define CW_INVALID_SESSION             (0x02U)
00083
00084 /* Key / path types for SIGN command (keyType) */
00085 #define CW_SIGN_CURR_K1                (0x00U)
00086 #define CW_SIGN_CURR_R1                (0x10U)
00087 #define CW_SIGN_DERIVE_K1              (0x01U)
00088 #define CW_SIGN_DERIVE_R1              (0x11U)
00089 #define CW_SIGN_PINLESS_K1             (0x03U)
00090
00091 /* PIN mode for SIGN command */
00092 #define CW_SIGN_WITH_PIN               (false)
00093 #define CW_SIGN_PINLESS                (true)
00094
00095 /* Signature types for SIGN command */
00096 #define CW_SIGN_SIG_ECDSA_LOW_S        (0x00U)
00097 #define CW_SIGN_SIG_ECDSA_EOSIO        (0x01U)
00098 #define CW_SIGN_SIG_SCHNORR_BIP340     (0x02U)
00099
00100 /* SIGN-specific error codes */
00101 #define CW_SIGN_KEY_TOO_SHORT                  (0x80U)
00102 #define CW_SIGN_NO_KEY_LOADED                  (0x81U)
00103 #define CW_SIGN_PIN_INCORRECT                  (0x82U)
00104 #define CW_SIGN_KEY_TOO_SHORT_WITH_PINLESS_MODE (0x83U)
```

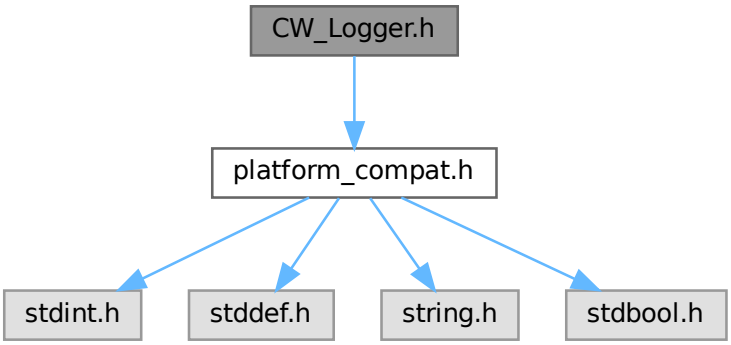
```
00105
00106 /* Size constants */
00107 #define CW_RAW_SIGNATURE_SIZE         (64U)
00108 #define CW_HASH_SIZE                  (32U)
00109 #define CW_MAX_DERIVE_PATH_LENGTH     (20U)
00110 #define CW_MIN_PIN_LENGTH              (4U)
00111 #define CW_MAX_PIN_LENGTH              (9U)
00112 #define CW_USER_DATA_PAGE_SIZE        (208U)
00113 #define CW_CONNECT_MAX_ATTEMPTS        (5U)
00114
00115 /* Byte offsets within a raw 64-byte signature (r[32] || s[32]) */
00116 #define CW_SIG_R_OFFSET               (0U)
00117 #define CW_SIG_S_OFFSET               (32U)
00118
00119 /* DER encoding tags (ASN.1) */
00120 #define CW_DER_TAG_SEQUENCE           (0x30U)
00121 #define CW_DER_TAG_INTEGER            (0x02U)
00122
00123 /* Certificate verification constants */
00124 #define CW_CERT_NONCE_SIZE            (8U)
00125
00126 /* Certificate verification result codes */
00127 #define CW_CERT_OK                    (0x00U)
00128 #define CW_CERT_FORMAT_ERROR          (0x10U)
00129 #define CW_CERT_NONCE_MISMATCH        (0x11U)
00130 #define CW_CERT_CARD_SIG_INVALID      (0x12U)
00131 #define CW_CERT_MANUF_SIG_INVALID     (0x13U)
00132 #define CW_CERT_KEY_NOT_FOUND         (0x14U)
00133
00134 /* Manufacturer certificate maximum buffer size (bytes).
00135  * Actual Cryptnox Basic G1 manufacturer certificate is 411 bytes (0x019B). */
00136 #define CW_MANUF_CERT_MAX_BYTES       (420U)
00137
00138 /****************************************************************
00139  * 3. CW_Curve enum
00140  ****************************************************************/
00141
00151 enum CW_Curve {
00152     CW_CURVE_SECP256R1 = 0,
00153     CW_CURVE_SECP256K1 = 1
00154 };
00155
00156 /****************************************************************
00157  * 4. CW_SecureSession struct
00158  ****************************************************************/
00159
00168 struct CW_SecureSession {
00169     uint8_t aesKey[CW_AESKEY_SIZE];
00170     uint8_t macKey[CW_MACKEY_SIZE];
00171     uint8_t iv[CW_IV_SIZE];
00172
00174     CW_SecureSession() {
00175         memset(aesKey, 0U, sizeof(aesKey));
00176         memset(macKey, 0U, sizeof(macKey));
00177         memset(iv, 0U, sizeof(iv));
00178     }
00179
00181     void clear() {
00182         CW_Utils::secure_wipe(aesKey, sizeof(aesKey));
00183         CW_Utils::secure_wipe(macKey, sizeof(macKey));
00184         CW_Utils::secure_wipe(iv,     sizeof(iv));
00185     }
00186 };
00187
00188 /****************************************************************
00189  * 5. Compile-time feature flags
00190  ****************************************************************/
00191
00195 #ifndef CW_VERIFY_CERT
00196 #define CW_VERIFY_CERT 1
00197 #endif
00198 #if CW_VERIFY_CERT == 0
00199 #  error "CW_VERIFY_CERT=0 disables certificate chain verification (CRIT-02/H-07). " \
00200          "Remove -DCW_VERIFY_CERT=0 from your build flags -- this gate must never be disabled."
00201 #endif
00202
00214 #ifndef CW_DEBUG_LOGGING
00215 #  define CW_DEBUG_LOGGING 0
00216 #endif
00217
00218 #if CW_DEBUG_LOGGING && defined(NDEBUG)
00219 #  error "CW_DEBUG_LOGGING=1 is set but NDEBUG is defined (release/optimised build). " \
00220          "Debug logging must not ship in production firmware -- it leaks session state " \
00221          "over UART.  Remove -DCW_DEBUG_LOGGING=1 from your release build flags (SEC-012)."
00222 #endif
00223
00224 #endif // CW_DEFS_H
```

4.9 CW_Logger.h File Reference

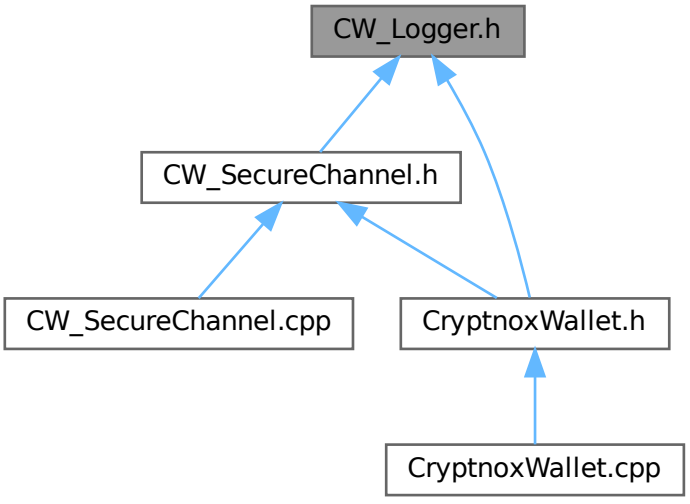
Abstract logging interface.

```
#include "platform_compat.h"
```

Include dependency graph for CW_Logger.h:



This graph shows which files directly or indirectly include this file:



Classes

- class CW_Logger

  Abstract interface for serial/debug output.

4.9.1 Detailed Description

Abstract logging interface.
Declares CW_Logger, the contract that any concrete output sink (UART, USB CDC, stdout, syslog, network, ...) must implement so the SDK remains independent of the host platform's logging facility.
The Arduino-style print/println overloads keep `F("...")`-quoted string literals in flash on Arduino targets; on non-Arduino targets `F()` is the identity macro (see platform_compat.h).
One of the three adapter interfaces a host integration must provide.
Definition in file CW_Logger.h.

4.10 CW_Logger.h

Go to the documentation of this file.

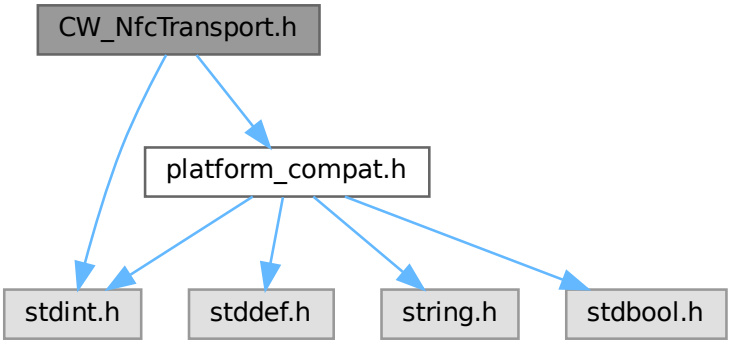
```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00020
00021 #ifndef CW_LOGGER_H
00022 #define CW_LOGGER_H
00023
00024 /**********************************************************************
00025  * 1. Included files
00026  **********************************************************************/
00027
00028 #include "platform_compat.h"
00029
00030 /**********************************************************************
00031  * 2. Class declaration
00032  **********************************************************************/
00033
00048 class CW_Logger {
00049 public:
00055     virtual bool begin(unsigned long baudRate = 115200UL) = 0;
00056
00059     virtual void print(const __FlashStringHelper* str) = 0;
00060     virtual void print(const char* str) = 0;
00061     virtual void print(char c) = 0;
00062     virtual void print(uint8_t value, int base = DEC) = 0;
00063     virtual void print(uint16_t value, int base = DEC) = 0;
00064     virtual void print(uint32_t value, int base = DEC) = 0;
00065     virtual void print(int value, int base = DEC) = 0;
00067
00070     virtual void println() = 0;
00071     virtual void println(const __FlashStringHelper* str) = 0;
00072     virtual void println(const char* str) = 0;
00073     virtual void println(char c) = 0;
00074     virtual void println(uint8_t value, int base = DEC) = 0;
00075     virtual void println(uint16_t value, int base = DEC) = 0;
00076     virtual void println(uint32_t value, int base = DEC) = 0;
00077     virtual void println(int value, int base = DEC) = 0;
00079
00080     virtual ~CW_Logger() {}
00081 };
00082
00083 #endif // CW_LOGGER_H
```

4.11 CW_NfcTransport.h File Reference

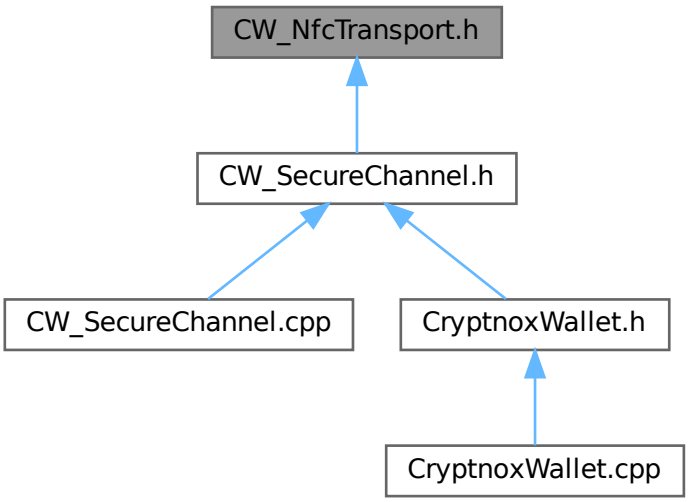
Abstract NFC transport interface.

```
#include "platform_compat.h"
#include <stdint.h>
```

Include dependency graph for CW_NfcTransport.h:



This graph shows which files directly or indirectly include this file:



Classes

- class CW_NfcTransport

  Abstract interface for NFC transport operations.

4.11.1 Detailed Description

Abstract NFC transport interface.
Declares CW_NfcTransport, the contract that any concrete NFC reader driver (PN532, PN7150, PC/↩ SC, . . .) must implement so that CW_SecureChannel and CryptnoxWallet remain hardware-agnostic. One of the three adapter interfaces a host integration must provide.

Definition in file CW_NfcTransport.h.

4.12 CW_NfcTransport.h

Go to the documentation of this file.

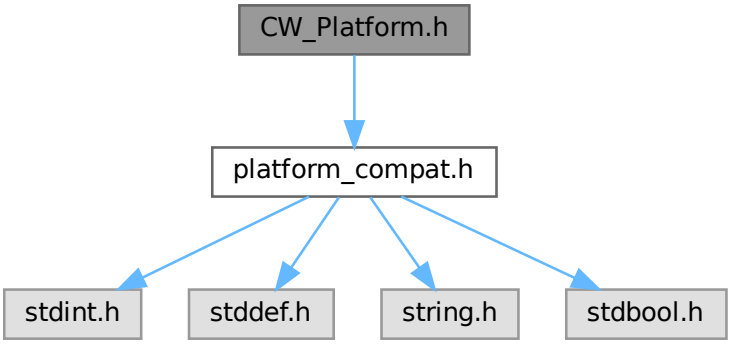
```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00016
00017 #ifndef CW_NFCTRANSPORT_H
00018 #define CW_NFCTRANSPORT_H
00019
00020 /**************************************************************************
00021  * 1. Included files
00022  *************************************************************************/
00023
00024 #include "platform_compat.h"
00025 #include <stdint.h>
00026
00027 /**************************************************************************
00028  * 2. Class declaration
00029  *************************************************************************/
00030
00040 class CW_NfcTransport {
00041 public:
00046     virtual bool begin() = 0;
00047
00052     virtual bool inListPassiveTarget() = 0;
00053
00063     virtual bool sendAPDU(const uint8_t* apdu, uint8_t apduLen,
00064                           uint8_t* response, uint8_t& responseLen) = 0;
00065
00081     virtual bool sendAPDULarge(const uint8_t* apdu, uint8_t apduLen,
00082                                uint8_t* response, uint16_t& responseLen) {
00083         uint8_t smallLen = static_cast<uint8_t>(
00084             (responseLen > static_cast<uint16_t>(UINT8_MAX))
00085                 ? static_cast<uint16_t>(UINT8_MAX)
00086                 : responseLen);
00087         bool result = sendAPDU(apdu, apduLen, response, smallLen);
00088         responseLen = static_cast<uint16_t>(smallLen);
00089         return result;
00090     }
00091
00095     virtual void resetReader() = 0;
00096
00101     virtual bool printFirmwareVersion() = 0;
00102
00103     virtual ~CW_NfcTransport() {}
00104 };
00105
00106 #endif // CW_NFCTRANSPORT_H
```

4.13 CW_Platform.h File Reference

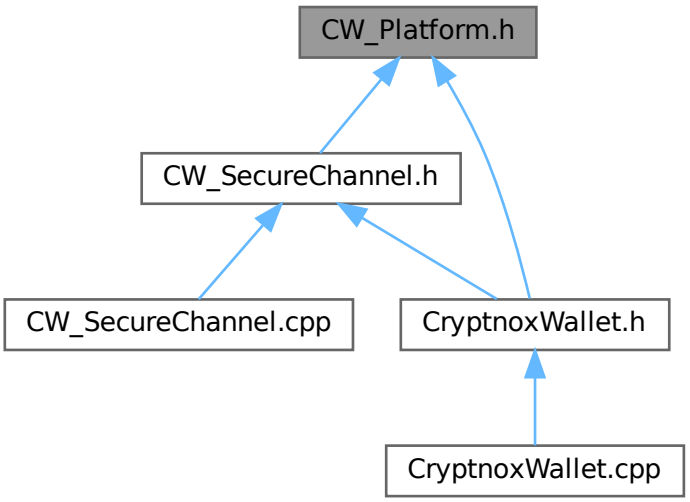
Abstract platform interface for timing primitives.

`#include "platform_compat.h"`

Include dependency graph for CW_Platform.h:



This graph shows which files directly or indirectly include this file:



Classes

- class CW_Platform

  Abstract interface for platform-specific operations used by the SDK.

4.13.1 Detailed Description

Abstract platform interface for timing primitives.
Declares CW_Platform, the contract that hosts implement so the SDK stays independent of any specific RTOS or bare-metal delay mechanism.

Currently exposes a single operation (CW_Platform::sleep_ms) used by CW_SecureChannel for inter-APDU spacing on slow NFC stacks.
Definition in file CW_Platform.h.

4.14 CW_Platform.h

Go to the documentation of this file.

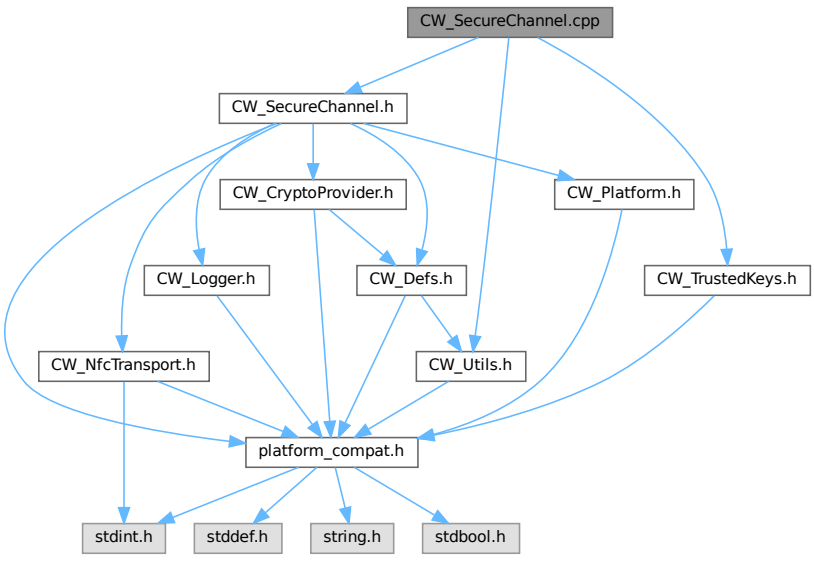
```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00016
00017 #ifndef CW_PLATFORM_H
00018 #define CW_PLATFORM_H
00019
00020 /**********************************************************************
00021  * 1. Included files
00022  **********************************************************************/
00023
00024 #include "platform_compat.h"
00025
00026 /**********************************************************************
00027  * 2. Class declaration
00028  **********************************************************************/
00029
00039 class CW_Platform {
00040 public:
00046     virtual void sleep_ms(uint32_t ms) = 0;
00047
00048     virtual ~CW_Platform() {}
00049 };
00050
00051 #endif /* CW_PLATFORM_H */
```

4.15 CW_SecureChannel.cpp File Reference

Implementation of the Cryptnox secure channel protocol.

```
#include "CW_SecureChannel.h"
#include "CW_Utils.h"
#include "CW_TrustedKeys.h"
```

Include dependency graph for CW_SecureChannel.cpp:



Macros

- #define RESPONSE_GETCARDCERTIFICATE_IN_BYTES 148U
- #define RESPONSE_SELECT_IN_BYTES 40U
- #define RESPONSE_GETMANUFACTURERCERT_PAGE_IN_BYTES 420U
- #define RESPONSE_OPENSECURECHANNEL_IN_BYTES 34U
- #define REQUEST_MUTUALLYAUTHENTICATE_IN_BYTES 69U
- #define RESPONSE_MUTUALLYAUTHENTICATE_IN_BYTES 66U
- #define RESPONSE_STATUS_WORDS_IN_BYTES 2U
- #define OPENSECURECHANNEL_SALT_IN_BYTES (RESPONSE_OPENSECURECHANNEL_IN_BYTES - RESPONSE_STATUS_WORDS_IN_BYTES)
- #define GETCARDCERTIFICATE_IN_BYTES (RESPONSE_GETCARDCERTIFICATE_IN_BYTES - RESPONSE_STATUS_WORDS_IN_BYTES)
- #define RANDOM_BYTES 8U
- #define COMMON_PAIRING_DATA CW_PAIRING_DATA
- #define CLIENT_PRIVATE_KEY_SIZE 32U
- #define CLIENT_PUBLIC_KEY_SIZE 64U
- #define CARDEPHEMERALPUBKEY_SIZE 64U
- #define AES_BLOCK_SIZE 16U
- #define APDU_HEADER_LEN (4U)
- #define APDU_LC_LEN (1U)
- #define MAC_APDU_LEN (12U)
- #define INPUT_BUFFER_LIMIT (CW_USER_DATA_PAGE_SIZE)
- #define ENC_BUF_MAX_LEN (INPUT_BUFFER_LIMIT + AES_BLOCK_SIZE)
- #define MAX_MAC_DATA_LEN (APDU_HEADER_LEN + MAC_APDU_LEN + ENC_BUF_MAX_LEN)
- #define SEND_APDU_MAX_LEN (APDU_HEADER_LEN + APDU_LC_LEN + AES_BLOCK_SIZE + ENC_BUF_MAX_LEN)
- #define DER_TAG_SEQUENCE (0x30U) /* SEQUENCE (universal, constructed) */
- #define DER_TAG_BIT_STRING (0x03U) /* BIT STRING */
- #define DER_TAG_CTX0 (0xA0U) /* [0] EXPLICIT — version in v3 TBSCertificate */
- #define DER_LEN_LONG_FLAG (0x80U) /* set = long-form length */
- #define DER_LEN_LONG_1 (0x81U) /* long form, 1 following byte */
- #define DER_LEN_LONG_2 (0x82U) /* long form, 2 following bytes */
- #define DER_EC_UNCOMPRESSED (0x04U) /* uncompressed point prefix */
- #define DER_EC_POINT_BYTES (65U) /* 0x04 || X[32] || Y[32] */
- #define DER_BIT_UNUSED_ZERO (0x00U) /* BIT STRING unused-bits field must be 0 */

Functions

- static bool derReadLength (const uint8_t *buf, uint16_t bufLen, uint16_t &pos, uint16_t &fieldLen)
- static bool derSkipField (const uint8_t *buf, uint16_t bufLen, uint16_t &pos)
- static bool derWalkMfCert (const uint8_t *buf, uint16_t bufLen, uint16_t &tbsMsgStart, uint16_t &tbsMsgLen, const uint8_t *&pubKey65Ptr, const uint8_t *&sigPtr, uint8_t &sigLen)

Variables

- static uint8_t s_apduBuf [SEND_APDU_MAX_LEN]
- static uint8_t s_macBuf [MAX_MAC_DATA_LEN]
- static uint8_t s_dataBuf [ENC_BUF_MAX_LEN]
- static uint8_t s_mfCertBuf [CW_MANUF_CERT_MAX_BYTES]

4.15.1 Detailed Description

Implementation of the Cryptnox secure channel protocol.
Implements the methods declared in CW_SecureChannel.h: APDU framing, certificate chain verification against the trusted CA keys (CW_TrustedKeys.h), ECDH session key derivation, AES-CBC encrypted messaging with rolling IV, and MAC verification on every response.
Module-level static scratch buffers are reused across calls to keep the stack footprint small; secret material is wiped after use.
Definition in file CW_SecureChannel.cpp.

4.15.2 Macro Definition Documentation

4.15.2.1 AES_BLOCK_SIZE

```
#define AES_BLOCK_SIZE 16U
```
Definition at line 45 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcDecrypt(), CW_SecureChannel::aesCbcEncrypt(), and CW_SecureChannel::mutuallyAuthenticate().

4.15.2.2 APDU_HEADER_LEN

```
#define APDU_HEADER_LEN (4U)
```
Definition at line 46 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::mutuallyAuthenticate().

4.15.2.3 APDU_LC_LEN

```
#define APDU_LC_LEN (1U)
```
Definition at line 47 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcEncrypt(), and CW_SecureChannel::mutuallyAuthenticate().

4.15.2.4 CARDEPHEMERALPUBKEY_SIZE

```
#define CARDEPHEMERALPUBKEY_SIZE 64U
```
Definition at line 44 of file CW_SecureChannel.cpp.

4.15.2.5 CLIENT_PRIVATE_KEY_SIZE

```
#define CLIENT_PRIVATE_KEY_SIZE 32U
```
Definition at line 42 of file CW_SecureChannel.cpp.

4.15.2.6 CLIENT_PUBLIC_KEY_SIZE

```
#define CLIENT_PUBLIC_KEY_SIZE 64U
```
Definition at line 43 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::openSecureChannel().

4.15.2.7 COMMON_PAIRING_DATA

```
#define COMMON_PAIRING_DATA CW_PAIRING_DATA
```
Definition at line 41 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::mutuallyAuthenticate().

4.15.2.8 DER_BIT_UNUSED_ZERO

```
#define DER_BIT_UNUSED_ZERO (0x00U) /* BIT STRING unused-bits field must be 0 */
```
Definition at line 80 of file CW_SecureChannel.cpp.
Referenced by derWalkMfCert().

4.15.2.9 DER_EC_POINT_BYTES

```
#define DER_EC_POINT_BYTES (65U) /* 0x04 || X[32] || Y[32] */
```
Definition at line 79 of file CW_SecureChannel.cpp.
Referenced by derWalkMfCert().

4.15.2.10 DER_EC_UNCOMPRESSED

```
#define DER_EC_UNCOMPRESSED (0x04U) /* uncompressed point prefix */
```
Definition at line 78 of file CW_SecureChannel.cpp.
Referenced by derWalkMfCert().

4.15.2.11 DER_LEN_LONG_1

```
#define DER_LEN_LONG_1 (0x81U) /* long form, 1 following byte */
```
Definition at line 74 of file CW_SecureChannel.cpp.
Referenced by derReadLength().

4.15.2.12 DER_LEN_LONG_2

```
#define DER_LEN_LONG_2 (0x82U) /* long form, 2 following bytes */
```
Definition at line 75 of file CW_SecureChannel.cpp.
Referenced by derReadLength().

4.15.2.13 DER_LEN_LONG_FLAG

```
#define DER_LEN_LONG_FLAG (0x80U) /* set = long-form length */
```
Definition at line 73 of file CW_SecureChannel.cpp.
Referenced by derReadLength().

4.15.2.14 DER_TAG_BIT_STRING

```
#define DER_TAG_BIT_STRING (0x03U) /* BIT STRING */
```
Definition at line 69 of file CW_SecureChannel.cpp.
Referenced by derWalkMfCert().

4.15.2.15 DER_TAG_CTX0

```
#define DER_TAG_CTX0 (0xA0U) /* [0] EXPLICIT -- version in v3 TBSCertificate */
```
Definition at line 70 of file CW_SecureChannel.cpp.
Referenced by derWalkMfCert().

4.15.2.16 DER_TAG_SEQUENCE

```
#define DER_TAG_SEQUENCE (0x30U) /* SEQUENCE (universal, constructed) */
```
Definition at line 68 of file CW_SecureChannel.cpp.
Referenced by derWalkMfCert().

4.15.2.17 ENC_BUF_MAX_LEN

```
#define ENC_BUF_MAX_LEN (INPUT_BUFFER_LIMIT + AES_BLOCK_SIZE)
```
Definition at line 50 of file CW_SecureChannel.cpp.

4.15.2.18 GETCARDCERTIFICATE_IN_BYTES

```
#define GETCARDCERTIFICATE_IN_BYTES (RESPONSE_GETCARDCERTIFICATE_IN_BYTES - RESPONSE_STATUS_WORDS_IN_BYTES)
```
Definition at line 38 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::getCardCertificate().

4.15.2.19 INPUT_BUFFER_LIMIT

```
#define INPUT_BUFFER_LIMIT (CW_USER_DATA_PAGE_SIZE)
```
Definition at line 49 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcEncrypt().

4.15.2.20 MAC_APDU_LEN

```
#define MAC_APDU_LEN (12U)
```
Definition at line 48 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcEncrypt().

4.15.2.21 MAX_MAC_DATA_LEN

```
#define MAX_MAC_DATA_LEN (APDU_HEADER_LEN + MAC_APDU_LEN + ENC_BUF_MAX_LEN)
```
Definition at line 51 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcEncrypt().

4.15.2.22 OPENSECURECHANNEL_SALT_IN_BYTES

```
#define OPENSECURECHANNEL_SALT_IN_BYTES (RESPONSE_OPENSECURECHANNEL_IN_BYTES - RESPONSE_STATUS_WORDS_IN_BYTES)
```
Definition at line 37 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::openSecureChannel().

4.15.2.23 RANDOM_BYTES

```
#define RANDOM_BYTES 8U
```
Definition at line 40 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::getCardCertificate().

4.15.2.24 REQUEST_MUTUALLYAUTHENTICATE_IN_BYTES

```
#define REQUEST_MUTUALLYAUTHENTICATE_IN_BYTES 69U
```
Definition at line 33 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::mutuallyAuthenticate().

4.15.2.25 RESPONSE_GETCARDCERTIFICATE_IN_BYTES

```
#define RESPONSE_GETCARDCERTIFICATE_IN_BYTES 148U
```
Definition at line 27 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::getCardCertificate().

4.15.2.26 RESPONSE_GETMANUFACTURERCERT_PAGE_IN_BYTES

```
#define RESPONSE_GETMANUFACTURERCERT_PAGE_IN_BYTES 420U
```
Definition at line 31 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::getManufacturerCertificate().

4.15.2.27 RESPONSE_MUTUALLYAUTHENTICATE_IN_BYTES

```
#define RESPONSE_MUTUALLYAUTHENTICATE_IN_BYTES 66U
```
Definition at line 34 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::mutuallyAuthenticate().

4.15.2.28 RESPONSE_OPENSECURECHANNEL_IN_BYTES

```
#define RESPONSE_OPENSECURECHANNEL_IN_BYTES 34U
```
Definition at line 32 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::openSecureChannel().

4.15.2.29 RESPONSE_SELECT_IN_BYTES

```
#define RESPONSE_SELECT_IN_BYTES 40U
```
Definition at line 29 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::selectApdu().

4.15.2.30 RESPONSE_STATUS_WORDS_IN_BYTES

```
#define RESPONSE_STATUS_WORDS_IN_BYTES 2U
```
Definition at line 35 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::getCardCertificate(), and CW_SecureChannel::getManufacturerCertificate().

4.15.2.31 SEND_APDU_MAX_LEN

```
#define SEND_APDU_MAX_LEN (APDU_HEADER_LEN + APDU_LC_LEN + AES_BLOCK_SIZE + ENC_BUF_MAX_LEN)
```
Definition at line 52 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcEncrypt().

4.15.3 Function Documentation

4.15.3.1 derReadLength()

```
bool derReadLength (
            const uint8_t * buf,
            uint16_t bufLen,
            uint16_t & pos,
            uint16_t & fieldLen)  [static]
```
Definition at line 732 of file CW_SecureChannel.cpp.
References DER_LEN_LONG_1, DER_LEN_LONG_2, and DER_LEN_LONG_FLAG.
Referenced by derSkipField(), and derWalkMfCert().

4.15.3.2 derSkipField()

```
bool derSkipField (
            const uint8_t * buf,
            uint16_t bufLen,
            uint16_t & pos)  [static]
```
Definition at line 767 of file CW_SecureChannel.cpp.
References derReadLength().
Referenced by derWalkMfCert().

4.15.3.3 derWalkMfCert()

```
bool derWalkMfCert (
            const uint8_t * buf,
            uint16_t bufLen,
            uint16_t & tbsMsgStart,
            uint16_t & tbsMsgLen,
            const uint8_t *& pubKey65Ptr,
            const uint8_t *& sigPtr,
            uint8_t & sigLen)  [static]
```
Definition at line 791 of file CW_SecureChannel.cpp.
References DER_BIT_UNUSED_ZERO, DER_EC_POINT_BYTES, DER_EC_UNCOMPRESSED, DER_TAG_BIT_STRING, DER_TAG_CTX0, DER_TAG_SEQUENCE, derReadLength(), and derSkipField().
Referenced by CW_SecureChannel::verifyCertificateChain().

4.15.4 Variable Documentation

4.15.4.1 s_apduBuf

```
uint8_t s_apduBuf[SEND_APDU_MAX_LEN]  [static]
```

Definition at line 60 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcDecrypt(), and CW_SecureChannel::aesCbcEncrypt().

4.15.4.2 s_dataBuf

```
uint8_t s_dataBuf[ENC_BUF_MAX_LEN]  [static]
```
Definition at line 62 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcDecrypt(), and CW_SecureChannel::aesCbcEncrypt().

4.15.4.3 s_macBuf

```
uint8_t s_macBuf[MAX_MAC_DATA_LEN]  [static]
```
Definition at line 61 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::aesCbcDecrypt(), and CW_SecureChannel::aesCbcEncrypt().

4.15.4.4 s_mfCertBuf

```
uint8_t s_mfCertBuf[CW_MANUF_CERT_MAX_BYTES]  [static]
```
Definition at line 65 of file CW_SecureChannel.cpp.
Referenced by CW_SecureChannel::preFetchManufacturerCert(), and CW_SecureChannel::verifyCertificateChain().

4.16 CW_SecureChannel.cpp

Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00018
00019 #include "CW_SecureChannel.h"
00020 #include "CW_Utils.h"
00021 #include "CW_TrustedKeys.h"
00022
00023 /*****************************************************************
00024  * Module-level constants
00025  *****************************************************************/
00026
00027 #define RESPONSE_GETCARDCERTIFICATE_IN_BYTES        148U
00028 /* SELECT AID: 1 (type) + 3 (ver) + 32 (status) + 2 (SW) = 38 bytes */
00029 #define RESPONSE_SELECT_IN_BYTES                     40U
00030 /* GET_MANUFACTURER_CERT: full DataOut up to certLen(2)+cert(411)+SW(2)=415 bytes; 420 for margin. */
00031 #define RESPONSE_GETMANUFACTURERCERT_PAGE_IN_BYTES  420U
00032 #define RESPONSE_OPENSECURECHANNEL_IN_BYTES      34U
00033 #define REQUEST_MUTUALLYAUTHENTICATE_IN_BYTES    69U
00034 #define RESPONSE_MUTUALLYAUTHENTICATE_IN_BYTES   66U
00035 #define RESPONSE_STATUS_WORDS_IN_BYTES            2U
00036
00037 #define OPENSECURECHANNEL_SALT_IN_BYTES   (RESPONSE_OPENSECURECHANNEL_IN_BYTES -
      RESPONSE_STATUS_WORDS_IN_BYTES)
00038 #define GETCARDCERTIFICATE_IN_BYTES       (RESPONSE_GETCARDCERTIFICATE_IN_BYTES -
      RESPONSE_STATUS_WORDS_IN_BYTES)
00039
00040 #define RANDOM_BYTES              8U
00041 #define COMMON_PAIRING_DATA       CW_PAIRING_DATA
00042 #define CLIENT_PRIVATE_KEY_SIZE  32U
00043 #define CLIENT_PUBLIC_KEY_SIZE   64U
00044 #define CARDEPHEMERALPUBKEY_SIZE 64U
00045 #define AES_BLOCK_SIZE           16U
00046 #define APDU_HEADER_LEN           (4U)
00047 #define APDU_LC_LEN               (1U)
00048 #define MAC_APDU_LEN             (12U)
00049 #define INPUT_BUFFER_LIMIT        (CW_USER_DATA_PAGE_SIZE)
00050 #define ENC_BUF_MAX_LEN           (INPUT_BUFFER_LIMIT + AES_BLOCK_SIZE)
00051 #define MAX_MAC_DATA_LEN          (APDU_HEADER_LEN + MAC_APDU_LEN + ENC_BUF_MAX_LEN)
00052 #define SEND_APDU_MAX_LEN         (APDU_HEADER_LEN + APDU_LC_LEN + AES_BLOCK_SIZE + ENC_BUF_MAX_LEN)
00053
00054 /* Enforce APDU fits within a single PN532 APDU (255 bytes max) */
00055 static_assert(APDU_HEADER_LEN + APDU_LC_LEN + AES_BLOCK_SIZE + ENC_BUF_MAX_LEN <= 255U,
00056               "CW_USER_DATA_PAGE_SIZE too large for PN532 single APDU transport");
00057
00058 /* Shared static crypto scratch buffers -- reuse is safe because decrypt is
00059  * always called from inside encrypt AFTER encrypt's large buffers are done. */
```

```
00060 static uint8_t s_apduBuf[SEND_APDU_MAX_LEN];  /* 245 bytes */
00061 static uint8_t s_macBuf [MAX_MAC_DATA_LEN];   /* 240 bytes */
00062 static uint8_t s_dataBuf[ENC_BUF_MAX_LEN];   /* 224 bytes */
00063
00064 /* Manufacturer certificate assembly buffer (used only during verifyCertificateChain). */
00065 static uint8_t s_mfCertBuf[CW_MANUF_CERT_MAX_BYTES];
00066
00067 /* DER TLV tag bytes */
00068 #define DER_TAG_SEQUENCE    (0x30U)  /* SEQUENCE (universal, constructed) */
00069 #define DER_TAG_BIT_STRING  (0x03U)  /* BIT STRING */
00070 #define DER_TAG_CTX0        (0xA0U)  /* [0] EXPLICIT -- version in v3 TBSCertificate */
00071
00072 /* DER length-field encoding */
00073 #define DER_LEN_LONG_FLAG   (0x80U)  /* set = long-form length */
00074 #define DER_LEN_LONG_1      (0x81U)  /* long form, 1 following byte */
00075 #define DER_LEN_LONG_2      (0x82U)  /* long form, 2 following bytes */
00076
00077 /* EC-point encoding */
00078 #define DER_EC_UNCOMPRESSED (0x04U)  /* uncompressed point prefix */
00079 #define DER_EC_POINT_BYTES  (65U)    /* 0x04 || X[32] || Y[32] */
00080 #define DER_BIT_UNUSED_ZERO (0x00U)  /* BIT STRING unused-bits field must be 0 */
00081
00082 /**************************************************************************
00083  * Constructor
00084  **************************************************************************/
00085
00086 // cppcheck-suppress misra-c2012-12.3 -- C++: member initializer-list commas are not the comma
      operator
00087 CW_SecureChannel::CW_SecureChannel(CW_NfcTransport& driver,
00088                                    CW_Logger& logger,
00089                                    CW_CryptoProvider& crypto,
00090                                    CW_Platform& platform)
00091     : _driver(driver), _logger(logger), _crypto(crypto), _platform(platform),
00092       _cachedMfCertLen(0U) {
00093     memset(_lastNonce, 0, sizeof(_lastNonce));
00094 }
00095
00096 /**************************************************************************
00097  * Transport delegation methods
00098  **************************************************************************/
00099
00100 bool CW_SecureChannel::begin() {
00101     return _driver.begin();
00102 }
00103
00104 bool CW_SecureChannel::inListPassiveTarget() {
00105     return _driver.inListPassiveTarget();
00106 }
00107
00108 void CW_SecureChannel::resetReader() {
00109     _driver.resetReader();
00110 }
00111
00112 bool CW_SecureChannel::printFirmwareVersion() {
00113     return _driver.printFirmwareVersion();
00114 }
00115
00116 /**************************************************************************
00117  * Private helpers
00118  **************************************************************************/
00119
00120 bool CW_SecureChannel::checkStatusWord(const uint8_t* response, uint16_t responseLength,
00121                                        uint8_t sw1Expected, uint8_t sw2Expected) {
00122     bool ret = false;
00123
00124     if ((response == NULL) || (responseLength < 2U)) {
00125 #if CW_DEBUG_LOGGING
00126         _logger.println(F("checkStatusWord: response too short."));
00127 #endif
00128     }
00129     else {
00130         uint8_t sw1 = response[responseLength - 2U];
00131         uint8_t sw2 = response[responseLength - 1U];
00132
00133         if ((sw1 == sw1Expected) && (sw2 == sw2Expected)) {
00134             ret = true;
00135         }
00136         else {
00137 #if CW_DEBUG_LOGGING
00138             _logger.print(F("SW: 0x"));
00139             if (sw1 < 16U) { _logger.print(F("0")); }
00140             _logger.print(sw1, HEX);
00141             _logger.print(F(" 0x"));
00142             if (sw2 < 16U) { _logger.print(F("0")); }
00143             _logger.println(sw2, HEX);
00144 #endif
00145         }
```

```
00146     }
00147
00148     return ret;
00149 }
00150
00151 /**************************************************************************
00152  * Public methods
00153  **************************************************************************/
00154
00155 bool CW_SecureChannel::selectApdu() {
00156     bool ret = false;
00157
00158     uint8_t selectApduCmd[] = {
00159         0x00, 0xA4, 0x04, 0x00,
00160         0x07,
00161         0xA0, 0x00, 0x00, 0x10, 0x00, 0x01, 0x12
00162     };
00163
00164     uint8_t response[RESPONSE_SELECT_IN_BYTES];
00165     uint8_t responseLength = static_cast<uint8_t>(sizeof(response));
00166
00167     if (_driver.sendAPDU(selectApduCmd, sizeof(selectApduCmd), response, responseLength)) {
00168         if (checkStatusWord(response, responseLength, 0x90U, 0x00U)) {
00169             ret = true;
00170         } else {
00171 #if CW_DEBUG_LOGGING
00172             _logger.println(F("Select APDU failed."));
00173 #endif
00174         }
00175     } else {
00176 #if CW_DEBUG_LOGGING
00177         _logger.println(F("APDU select failed."));
00178 #endif
00179     }
00180
00181     return ret;
00182 }
00183
00184 bool CW_SecureChannel::getCardCertificate(uint8_t* cardCertificate, uint8_t& cardCertificateLength) {
00185     bool ret = false;
00186     uint8_t getCardCertificateResponse[RESPONSE_GETCARDCERTIFICATE_IN_BYTES];
00187     uint8_t getCardCertificateResponseLength =
      static_cast<uint8_t>(sizeof(getCardCertificateResponse));
00188
00189     if (cardCertificate != NULL) {
00190         uint8_t randomBytes[RANDOM_BYTES] = { 0U };
00191         if (!_crypto.random(randomBytes, RANDOM_BYTES)) {
00192 #if CW_DEBUG_LOGGING
00193             _logger.println(F("getCardCertificate: RNG failed."));
00194 #endif
00195             return false;
00196         }
00197
00198         /* Store nonce for replay check in verifyCertificateChain(). */
00199         (void)CW_Utils::safe_memcpy(_lastNonce, RANDOM_BYTES, randomBytes, RANDOM_BYTES);
00200
00201         uint8_t getCardCertificateApdu[] = {
00202             0x80, 0xF8, 0x00, 0x00, 0x08
00203         };
00204
00205         uint8_t fullApdu[sizeof(getCardCertificateApdu) + RANDOM_BYTES];
00206         (void)CW_Utils::safe_memcpy(fullApdu, sizeof(fullApdu), getCardCertificateApdu,
      sizeof(getCardCertificateApdu));
00207         (void)CW_Utils::safe_memcpy(fullApdu + sizeof(getCardCertificateApdu), RANDOM_BYTES,
      randomBytes, RANDOM_BYTES);
00208
00209         if (_driver.sendAPDU(fullApdu, sizeof(fullApdu),
00210                              getCardCertificateResponse, getCardCertificateResponseLength)) {
00211             if (checkStatusWord(getCardCertificateResponse, getCardCertificateResponseLength,
00212                                 0x90U, 0x00U)) {
00213                 cardCertificateLength = static_cast<uint8_t>(
00214                     static_cast<uint8_t>(getCardCertificateResponseLength -
      RESPONSE_STATUS_WORDS_IN_BYTES));
00215                 (void)CW_Utils::safe_memcpy(cardCertificate, GETCARDCERTIFICATE_IN_BYTES,
      getCardCertificateResponse, cardCertificateLength);
00216                 ret = true;
00217             } else {
00218 #if CW_DEBUG_LOGGING
00219                 _logger.println(F("getCardCertificate: bad SW."));
00220 #endif
00221             }
00222         } else {
00223 #if CW_DEBUG_LOGGING
00224             _logger.println(F("getCardCertificate APDU failed."));
00225 #endif
00226         }
00227     }
```

```
00228
00229     return ret;
00230 }
00231
00232 bool CW_SecureChannel::extractCardEphemeralKey(const uint8_t* cardCertificate,
00233                                                uint8_t* cardEphemeralPubKey,
00234                                                uint8_t* fullEphemeralPubKey65) {
00235     bool ret = false;
00236
00237     if ((cardCertificate == NULL) || (cardEphemeralPubKey == NULL)) {
00238         ret = false;
00239     }
00240     else {
00241         const uint8_t keyStart = 1U + 8U; /* skip 'C' and nonce */
00242         const uint8_t fullKeyLength = 65U;
00243
00244         /* Reject any key that is not an uncompressed point (0x04 prefix). */
00245         if (cardCertificate[keyStart] != 0x04U) {
00246             ret = false;
00247         }
00248         else {
00249             for (uint8_t i = 0U; i < fullKeyLength; i++) {
00250                 uint8_t b = cardCertificate[keyStart + i];
00251                 if (fullEphemeralPubKey65 != NULL) {
00252                     fullEphemeralPubKey65[i] = b;
00253                 }
00254                 if (i > 0U) {
00255                     cardEphemeralPubKey[i - 1U] = b;
00256                 }
00257             }
00258             ret = true;
00259         }
00260     }
00261
00262     return ret;
00263 }
00264
00265 bool CW_SecureChannel::openSecureChannel(uint8_t* salt,
00266                                          uint8_t* sessionPublicKey,
00267                                          uint8_t* sessionPrivateKey,
00268                                          CW_Curve sessionCurve) {
00269     bool ret = false;
00270
00271     if (!_crypto.makeKey(sessionPublicKey, sessionPrivateKey, sessionCurve)) {
00272 #if CW_DEBUG_LOGGING
00273         _logger.println(F("ECC key generation failed."));
00274 #endif
00275     }
00276     else {
00277         uint8_t opcApduHeader[] = {
00278             0x80, 0x10, 0x00, 0x00, 0x41, 0x04
00279         };
00280
00281         uint8_t fullApdu[sizeof(opcApduHeader) + CLIENT_PUBLIC_KEY_SIZE];
00282         (void)CW_Utils::safe_memcpy(fullApdu, sizeof(fullApdu), opcApduHeader, sizeof(opcApduHeader));
00283         (void)CW_Utils::safe_memcpy(fullApdu + sizeof(opcApduHeader), CLIENT_PUBLIC_KEY_SIZE,
    sessionPublicKey, CLIENT_PUBLIC_KEY_SIZE);
00284
00285         uint8_t response[RESPONSE_OPENSECURECHANNEL_IN_BYTES];
00286         uint8_t responseLength = static_cast<uint8_t>(sizeof(response));
00287
00288         if (_driver.sendAPDU(fullApdu, sizeof(fullApdu), response, responseLength)) {
00289             if (checkStatusWord(response, responseLength, 0x90U, 0x00U)) {
00290                 if (responseLength == static_cast<uint8_t>(RESPONSE_OPENSECURECHANNEL_IN_BYTES)) {
00291                     (void)CW_Utils::safe_memcpy(salt, OPENSECURECHANNEL_SALT_IN_BYTES, response,
    OPENSECURECHANNEL_SALT_IN_BYTES);
00292                     ret = true;
00293                 } else {
00294 #if CW_DEBUG_LOGGING
00295                     _logger.println(F("OpenSecureChannel: unexpected response size."));
00296 #endif
00297                 }
00298             } else {
00299 #if CW_DEBUG_LOGGING
00300                 _logger.println(F("OpenSecureChannel: bad SW."));
00301 #endif
00302             }
00303         } else {
00304 #if CW_DEBUG_LOGGING
00305             _logger.println(F("OpenSecureChannel APDU failed."));
00306 #endif
00307         }
00308     }
00309
00310     return ret;
00311 }
00312
```

```
00334 bool CW_SecureChannel::mutuallyAuthenticate(CW_SecureSession& session,
00335                                             const uint8_t* salt,
00336                                             uint8_t* clientPublicKey,
00337                                             const uint8_t* clientPrivateKey,
00338                                             CW_Curve sessionCurve,
00339                                             const uint8_t* cardEphemeralPubKey) {
00340     bool ret = false;
00341     uint8_t sharedSecret[32U] = { 0U };
00342     (void)clientPublicKey;
00343 
00344     if (!_crypto.ecdh(cardEphemeralPubKey, clientPrivateKey, sharedSecret, sessionCurve)) {
00345 #if CW_DEBUG_LOGGING
00346         _logger.println(F("ECDH failed."));
00347 #endif
00348     }
00349     else {
00350         uint8_t concat[32U + sizeof(COMMON_PAIRING_DATA) - 1U + 32U] = { 0U };
00351         uint8_t sha512Output[64U] = { 0U };
00352         const size_t pairingKeyLen = sizeof(COMMON_PAIRING_DATA) - 1U;
00353         const size_t concatLen     = 32U + pairingKeyLen + 32U;
00354 
00355         (void)CW_Utils::safe_memcpy(concat, sizeof(concat), sharedSecret, 32U);
00356         (void)CW_Utils::safe_memcpy(concat + 32U, sizeof(concat) - 32U, reinterpret_cast<const
      uint8_t*>(COMMON_PAIRING_DATA), pairingKeyLen);
00357         (void)CW_Utils::safe_memcpy(concat + 32U + pairingKeyLen, 32U, salt, 32U);
00358 
00359         bool sha512Ok = _crypto.sha512(concat, concatLen, sha512Output);
00360 
00361         if (!sha512Ok) {
00362             CW_Utils::secure_wipe(sharedSecret, sizeof(sharedSecret));
00363             CW_Utils::secure_wipe(sha512Output, sizeof(sha512Output));
00364             CW_Utils::secure_wipe(concat, sizeof(concat));
00365             return false;
00366         }
00367 
00368         (void)CW_Utils::safe_memcpy(session.aesKey, CW_AESKEY_SIZE, sha512Output, CW_AESKEY_SIZE);
00369         (void)CW_Utils::safe_memcpy(session.macKey, CW_MACKEY_SIZE, sha512Output + CW_AESKEY_SIZE,
      CW_MACKEY_SIZE);
00370 
00371         uint8_t iv_opc[AES_BLOCK_SIZE] = { 0U };
00372         uint8_t mac_iv[AES_BLOCK_SIZE] = { 0U };
00373         memset(iv_opc, 0x01U, AES_BLOCK_SIZE);
00374 
00375         uint8_t RNG_data[32U] = { 0U };
00376         if (!_crypto.random(RNG_data, sizeof(RNG_data))) {
00377 #if CW_DEBUG_LOGGING
00378             _logger.println(F("RNG failed."));
00379 #endif
00380             session.clear();
00381             CW_Utils::secure_wipe(sharedSecret, sizeof(sharedSecret));
00382             CW_Utils::secure_wipe(sha512Output, sizeof(sha512Output));
00383             CW_Utils::secure_wipe(concat, sizeof(concat));
00384             return false;
00385         }
00386 
00387         /* Encrypt random data with Kenc (Bit padding) */
00388         uint8_t ciphertextOPC[48U] = { 0U };
00389         uint16_t cipherLength = _crypto.aesCbcEncrypt(RNG_data, sizeof(RNG_data),
00390                                                       ciphertextOPC,
00391                                                       session.aesKey, sizeof(session.aesKey),
00392                                                       iv_opc, true);
00393 
00394         /* Compute MAC over APDU header + ciphertext (Null padding) */
00395         uint8_t opcApduHeader[APDU_HEADER_LEN + APDU_LC_LEN] = {
00396             0x80U, 0x11U, 0x00U, 0x00U,
00397             (uint8_t)(cipherLength + AES_BLOCK_SIZE)
00398         };
00399         uint8_t MAC_apduHeader[AES_BLOCK_SIZE] = { 0U };
00400         (void)CW_Utils::safe_memcpy(MAC_apduHeader, sizeof(MAC_apduHeader), opcApduHeader,
      sizeof(opcApduHeader));
00401 
00402         size_t  MAC_data_length = sizeof(MAC_apduHeader) + cipherLength;
00403         uint8_t MAC_data[64U] = { 0U };
00404         uint8_t ciphertextMACLong[64U] = { 0U };
00405 
00406         if (MAC_data_length > sizeof(MAC_data)) {
00407             session.clear();
00408             CW_Utils::secure_wipe(sharedSecret, sizeof(sharedSecret));
00409             CW_Utils::secure_wipe(sha512Output, sizeof(sha512Output));
00410             CW_Utils::secure_wipe(concat, sizeof(concat));
00411             CW_Utils::secure_wipe(RNG_data, sizeof(RNG_data));
00412             return false;
00413         }
00414 
00415         (void)CW_Utils::safe_memcpy(MAC_data, sizeof(MAC_data), MAC_apduHeader,
      sizeof(MAC_apduHeader));
00416         (void)CW_Utils::safe_memcpy(MAC_data + sizeof(MAC_apduHeader), sizeof(MAC_data) -
```

```
    sizeof(MAC_apduHeader), ciphertextOPC, cipherLength);
00417
00418         uint16_t encryptedLengthMAC = _crypto.aesCbcEncrypt(MAC_data, (uint16_t)MAC_data_length,
00419                                                             ciphertextMACLong,
00420                                                             session.macKey, sizeof(session.macKey),
00421                                                             mac_iv, false);
00422
00423         uint8_t MAC_value[AES_BLOCK_SIZE] = { 0U };
00424         uint8_t macOffset = (uint8_t)(encryptedLengthMAC - AES_BLOCK_SIZE);
00425         (void)CW_Utils::safe_memcpy(MAC_value, sizeof(MAC_value), ciphertextMACLong + macOffset,
    AES_BLOCK_SIZE);
00426
00427         /* Forge MUTUALLY AUTHENTICATE APDU */
00428         uint8_t sendApduOpc[REQUEST_MUTUALLYAUTHENTICATE_IN_BYTES] = { 0U };
00429         uint16_t offset = 0U;
00430         (void)CW_Utils::safe_memcpy(sendApduOpc + offset, sizeof(sendApduOpc) -
    static_cast<size_t>(offset), opcApduHeader, sizeof(opcApduHeader));
00431         offset += sizeof(opcApduHeader);
00432         (void)CW_Utils::safe_memcpy(sendApduOpc + offset, sizeof(sendApduOpc) -
    static_cast<size_t>(offset), MAC_value, sizeof(MAC_value));
00433         offset += sizeof(MAC_value);
00434         (void)CW_Utils::safe_memcpy(sendApduOpc + offset, sizeof(sendApduOpc) -
    static_cast<size_t>(offset), ciphertextOPC, cipherLength);
00435
00436         uint8_t response[255U] = { 0U };
00437         uint8_t responseLength = static_cast<uint8_t>(sizeof(response));
00438
00439         if (_driver.sendAPDU(sendApduOpc, sizeof(sendApduOpc), response, responseLength)) {
00440             if (checkStatusWord(response, responseLength, 0x90U, 0x00U)) {
00441                 if (responseLength == static_cast<uint8_t>(RESPONSE_MUTUALLYAUTHENTICATE_IN_BYTES)) {
00442                     (void)CW_Utils::safe_memcpy(session.iv, CW_IV_SIZE, response, CW_IV_SIZE);
00443                     ret = true;
00444                 } else {
00445 #if CW_DEBUG_LOGGING
00446                     _logger.println(F("MutualAuth: unexpected response size."));
00447 #endif
00448                 }
00449             } else {
00450 #if CW_DEBUG_LOGGING
00451                 _logger.println(F("MutualAuth: bad SW."));
00452 #endif
00453             }
00454         } else {
00455 #if CW_DEBUG_LOGGING
00456             _logger.println(F("MutualAuth APDU failed."));
00457 #endif
00458         }
00459
00460         /* Secure cleanup */
00461         CW_Utils::secure_wipe(sharedSecret, sizeof(sharedSecret));
00462         CW_Utils::secure_wipe(sha512Output, sizeof(sha512Output));
00463         CW_Utils::secure_wipe(concat, sizeof(concat));
00464         CW_Utils::secure_wipe(RNG_data, sizeof(RNG_data));
00465         CW_Utils::secure_wipe(ciphertextOPC, sizeof(ciphertextOPC));
00466         CW_Utils::secure_wipe(MAC_data, sizeof(MAC_data));
00467
00468         /* If the APDU exchange failed after session keys were written, clear
00469          * them now to prevent a half-initialised session from being used (CRIT-04). */
00470         if (!ret) {
00471             session.clear();
00472         }
00473     }
00474
00475     return ret;
00476 }
00477
00501 bool CW_SecureChannel::aesCbcEncrypt(CW_SecureSession& session,
00502                                      const uint8_t apdu[], uint16_t apduLength,
00503                                      const uint8_t data[], uint16_t dataLength,
00504                                      uint8_t* decryptedOutput, uint16_t* decryptedOutputLength) {
00505     bool ret = false;
00506
00507     /* Reject payloads that would overflow s_dataBuf (MED-01). */
00508     if (dataLength > INPUT_BUFFER_LIMIT) {
00509 #if CW_DEBUG_LOGGING
00510         _logger.println(F("Error: data too large for encryption buffer."));
00511 #endif
00512         return false;
00513     }
00514
00515     /* 1. Encrypt data with Kenc (Bit padding) */
00516     uint16_t encryptedLength = _crypto.aesCbcEncrypt(data, dataLength, s_dataBuf,
00517                                                      session.aesKey, sizeof(session.aesKey),
00518                                                      session.iv, true);
00519
00520     uint16_t lcValue = encryptedLength + (uint16_t)AES_BLOCK_SIZE;
00521     /* lcValue must fit in uint8_t: with INPUT_BUFFER_LIMIT=208, max encryptedLength=224,
```

```
00522      * so max lcValue=240. The static_assert above also caps APDU at 255 bytes (MED-03). */
00523     if (lcValue > 0xFFU) {
00524 #if CW_DEBUG_LOGGING
00525         _logger.println(F("Error: lcValue overflow -- payload too large."));
00526 #endif
00527         CW_Utils::secure_wipe(s_dataBuf, sizeof(s_dataBuf));
00528         return false;
00529     }
00530     uint8_t macApdu[MAC_APDU_LEN] = { 0U };
00531     /* MED-03: Single-byte length encoding and no direction byte are intentional --
00532      * this CBC-MAC construction matches the Cryptnox SCCP card firmware spec.
00533      * Changing to AES-CMAC, wider encoding, or adding a direction byte would
00534      * break the card protocol. lcValue overflow is prevented by the
00535      * INPUT_BUFFER_LIMIT precondition above (dataLength <= 208 → lcValue <= 240). */
00536     macApdu[0U] = (uint8_t)lcValue;
00537
00538     uint16_t macDataLength = apduLength + sizeof(macApdu) + encryptedLength;
00539     if (macDataLength > MAX_MAC_DATA_LEN) {
00540 #if CW_DEBUG_LOGGING
00541         _logger.println(F("Error: MAC data length exceeds buffer."));
00542 #endif
00543         CW_Utils::secure_wipe(s_dataBuf, sizeof(s_dataBuf));
00544         return false;
00545     }
00546
00547     /* 2. Build MAC input: APDU header || LC block || ciphertext */
00548     uint16_t offset = 0U;
00549     (void)CW_Utils::safe_memcpy(s_macBuf, sizeof(s_macBuf), apdu, apduLength);
00550     offset += apduLength;
00551     (void)CW_Utils::safe_memcpy(s_macBuf + offset, sizeof(s_macBuf) - static_cast<size_t>(offset),
      macApdu, sizeof(macApdu));
00552     offset += sizeof(macApdu);
00553     (void)CW_Utils::safe_memcpy(s_macBuf + offset, sizeof(s_macBuf) - static_cast<size_t>(offset),
      s_dataBuf, encryptedLength);
00554
00555     uint8_t macIv[AES_BLOCK_SIZE] = { 0U };
00556     uint16_t macEncryptedLength = _crypto.aesCbcEncrypt(s_macBuf, macDataLength, s_apduBuf,
00557                                                         session.macKey, sizeof(session.macKey),
00558                                                         macIv, false);
00559
00560     uint8_t macValue[AES_BLOCK_SIZE] = { 0U };
00561     uint16_t macOffset = macEncryptedLength - AES_BLOCK_SIZE;
00562     (void)CW_Utils::safe_memcpy(macValue, sizeof(macValue), s_apduBuf + macOffset, AES_BLOCK_SIZE);
00563
00564     /* 3. Build send APDU: header || Lc || MAC || ciphertext */
00565     const uint8_t lc = (uint8_t)lcValue;
00566     uint8_t sendApduLength = (uint8_t)(apduLength + APDU_LC_LEN + sizeof(macValue) + encryptedLength);
00567     if (sendApduLength > SEND_APDU_MAX_LEN) {
00568 #if CW_DEBUG_LOGGING
00569         _logger.println(F("Error: Send APDU length exceeds buffer."));
00570 #endif
00571         CW_Utils::secure_wipe(s_dataBuf, sizeof(s_dataBuf));
00572         CW_Utils::secure_wipe(s_macBuf,  sizeof(s_macBuf));
00573         return false;
00574     }
00575
00576     offset = 0U;
00577     (void)CW_Utils::safe_memcpy(s_apduBuf, sizeof(s_apduBuf), apdu, apduLength);
00578     offset += apduLength;
00579     s_apduBuf[offset] = lc;
00580     offset += APDU_LC_LEN;
00581     (void)CW_Utils::safe_memcpy(s_apduBuf + offset, sizeof(s_apduBuf) - static_cast<size_t>(offset),
      macValue, sizeof(macValue));
00582     offset += sizeof(macValue);
00583     (void)CW_Utils::safe_memcpy(s_apduBuf + offset, sizeof(s_apduBuf) - static_cast<size_t>(offset),
      s_dataBuf, encryptedLength);
00584
00585     /* 4. Send APDU */
00586     uint8_t response[255U] = { 0U };
00587     uint8_t responseLength = static_cast<uint8_t>(sizeof(response));
00588
00589     if (_driver.sendAPDU(s_apduBuf, sendApduLength, response, responseLength)) {
00590         if (checkStatusWord(response, responseLength, 0x90U, 0x00U)) {
00591             /* Update session.iv ONLY after the response MAC is verified (HIGH-01).
00592              * Moving it before aesCbcDecrypt would let an attacker-chosen IV
00593              * desynchronise the rolling-IV state even on MAC failure. */
00594             ret = aesCbcDecrypt(session, response, static_cast<size_t>(responseLength), macValue,
00595                                 decryptedOutput, decryptedOutputLength);
00596             if (ret) {
00597                 (void)CW_Utils::safe_memcpy(session.iv, CW_IV_SIZE, response, CW_IV_SIZE);
00598             } else {
00599                 session.clear();
00600             }
00601         } else if (responseLength >= 2U) {
00602             /* Card-level error: surface the SW so the caller can diagnose
00603              * common cases (e.g. 0x63Cn = wrong PIN with n retries left). */
00604 #if CW_DEBUG_LOGGING
```

```
00605             _logger.print(F("Secured APDU: bad SW 0x"));
00606             if (response[responseLength - 2U] < 0x10U) { _logger.print(F("0")); }
00607             _logger.print(response[responseLength - 2U], HEX);
00608             if (response[responseLength - 1U] < 0x10U) { _logger.print(F("0")); }
00609             _logger.println(response[responseLength - 1U], HEX);
00610 #endif
00611         }
00612     } else {
00613 #if CW_DEBUG_LOGGING
00614         _logger.println(F("Secured APDU failed."));
00615 #endif
00616     }
00617
00618     /* Wipe plaintext scratch buffers so they do not persist in .bss (MED-04). */
00619     CW_Utils::secure_wipe(s_dataBuf, sizeof(s_dataBuf));
00620     CW_Utils::secure_wipe(s_macBuf,  sizeof(s_macBuf));
00621
00622     return ret;
00623 }
00624
00625 bool CW_SecureChannel::aesCbcDecrypt(const CW_SecureSession& session,
00626                                      uint8_t* response, size_t response_len,
00627                                      uint8_t* mac_value,
00628                                      uint8_t* decryptedOutput, uint16_t* decryptedOutputLength) {
00629     /* Precondition: response must hold at least MAC(16) + 1 ciphertext byte + SW(2) (HIGH-02).
00630      * Without this check, response_len < 18 causes size_t underflow in the subtractions below. */
00631     if ((response == NULL) || (response_len < (size_t)(AES_BLOCK_SIZE + 2U + 1U))) {
00632         return false;
00633     }
00634
00635     /* Response layout: MAC(16) || cipherText(N) || SW1(1) || SW2(1) */
00636     uint8_t rep_mac[AES_BLOCK_SIZE];
00637     (void)CW_Utils::safe_memcpy(rep_mac, sizeof(rep_mac), response, AES_BLOCK_SIZE);
00638     uint8_t* rep_data  = response + AES_BLOCK_SIZE;
00639     size_t totalDataLen = response_len - 2U;
00640     size_t cipherLen    = totalDataLen - AES_BLOCK_SIZE;
00641
00642     if (mac_value == NULL) {
00643         return false;
00644     }
00645
00646     /* Verify MAC: AES-CBC-MAC over [length_header(16)] || [all_ciphertext] */
00647     size_t macInputLen = AES_BLOCK_SIZE + cipherLen;
00648     if (macInputLen > sizeof(s_macBuf)) {
00649 #if CW_DEBUG_LOGGING
00650         _logger.println(F("Error: Response too large for MAC verification."));
00651 #endif
00652         return false;
00653     }
00654
00655     memset(s_macBuf, 0U, AES_BLOCK_SIZE);
00656     s_macBuf[0] = (uint8_t)totalDataLen;
00657     (void)CW_Utils::safe_memcpy(s_macBuf + AES_BLOCK_SIZE, sizeof(s_macBuf) - AES_BLOCK_SIZE,
      rep_data, cipherLen);
00658
00659     uint8_t mac_iv[AES_BLOCK_SIZE] = { 0U };
00660     uint16_t macEncryptedLength = _crypto.aesCbcEncrypt(s_macBuf, (uint16_t)macInputLen, s_apduBuf,
00661                                                         session.macKey, sizeof(session.macKey),
00662                                                         mac_iv, false);
00663
00664     uint8_t recomputedMacValue[AES_BLOCK_SIZE] = { 0U };
00665     uint16_t macOffset = macEncryptedLength - AES_BLOCK_SIZE;
00666     (void)CW_Utils::safe_memcpy(recomputedMacValue, sizeof(recomputedMacValue), s_apduBuf + macOffset,
      AES_BLOCK_SIZE);
00667
00668     if (!CW_Utils::secure_compare(rep_mac, recomputedMacValue, AES_BLOCK_SIZE)) {
00669 #if CW_DEBUG_LOGGING
00670         _logger.println(F("MAC mismatch."));
00671 #endif
00672         CW_Utils::secure_wipe(s_macBuf, sizeof(s_macBuf));
00673         return false;
00674     }
00675
00676     /* Decrypt ciphertext using mac_value as IV (Bit padding removal) */
00677     uint16_t decryptedDataLength = _crypto.aesCbcDecrypt(rep_data, (uint16_t)cipherLen, s_dataBuf,
00678                                                          session.aesKey, sizeof(session.aesKey),
00679                                                          mac_value, true);
00680
00681     bool ret = false;
00682
00683     if (decryptedDataLength < 2U) {
00684 #if CW_DEBUG_LOGGING
00685         _logger.println(F("Error: Decoded data too short."));
00686 #endif
00687     }
00688     else if (decryptedDataLength > sizeof(s_dataBuf)) {
00689 #if CW_DEBUG_LOGGING
```

```
00690         _logger.println(F("Error: Decoded data length exceeds buffer."));
00691 #endif
00692     }
00693     else {
00694         uint8_t innerSW1 = s_dataBuf[decryptedDataLength - 2U];
00695         uint8_t innerSW2 = s_dataBuf[decryptedDataLength - 1U];
00696         uint16_t payloadLength = decryptedDataLength - 2U;
00697 
00698         if ((innerSW1 != 0x90U) || (innerSW2 != 0x00U)) {
00699 #if CW_DEBUG_LOGGING
00700             _logger.print(F("Card error SW: 0x"));
00701             if (innerSW1 < 0x10U) { _logger.print(F("0")); }
00702             _logger.print(innerSW1, HEX);
00703             _logger.print(F(" 0x"));
00704             if (innerSW2 < 0x10U) { _logger.print(F("0")); }
00705             _logger.println(innerSW2, HEX);
00706 #endif
00707         }
00708         else {
00709             ret = true;
00710         }
00711 
00712         if ((decryptedOutput != NULL) && (decryptedOutputLength != NULL)) {
00713             (void)CW_Utils::safe_memcpy(decryptedOutput, sizeof(s_dataBuf), s_dataBuf, payloadLength);
00714             *decryptedOutputLength = payloadLength;
00715         }
00716     }
00717 
00718     /* Wipe plaintext scratch buffers (MED-04). */
00719     CW_Utils::secure_wipe(s_dataBuf, sizeof(s_dataBuf));
00720     CW_Utils::secure_wipe(s_macBuf,  sizeof(s_macBuf));
00721 
00722     return ret;
00723 }
00724 
00725 /**************************************************************
00726  * Certificate verification -- static helpers
00727  **************************************************************/
00728 
00729 
00730 /* Read the DER length at buf[*pos]; advance *pos past the length bytes.
00731  * Supports short form and long form with 1 or 2 extra bytes only. */
00732 static bool derReadLength(const uint8_t* buf, uint16_t bufLen,
00733                           uint16_t& pos, uint16_t& fieldLen) {
00734     bool    ok    = false;
00735     fieldLen      = 0U;
00736 
00737     if ((buf != NULL) && (pos < bufLen)) {
00738         uint8_t b = buf[pos];
00739         pos += 1U;
00740 
00741         if ((b & DER_LEN_LONG_FLAG) == 0U) {
00742             fieldLen = static_cast<uint16_t>(b);
00743             ok = true;
00744         } else if (b == DER_LEN_LONG_1) {
00745             if (pos < bufLen) {
00746                 fieldLen = static_cast<uint16_t>(buf[pos]);
00747                 pos += 1U;
00748                 ok = true;
00749             }
00750         } else if (b == DER_LEN_LONG_2) {
00751             if ((pos + 1U) < bufLen) {
00752                 fieldLen  = static_cast<uint16_t>(
00753                                 static_cast<uint16_t>(buf[pos]) << 8U);
00754                 fieldLen |= static_cast<uint16_t>(buf[pos + 1U]);
00755                 pos += 2U;
00756                 ok = true;
00757             }
00758         } else {
00759             ok = false; /* indefinite form or > 2 extra bytes -- unsupported */
00760         }
00761     }
00762 
00763     return ok;
00764 }
00765 
00766 /* Skip one complete DER TLV (tag byte + length bytes + value) at buf[*pos]. */
00767 static bool derSkipField(const uint8_t* buf, uint16_t bufLen, uint16_t& pos) {
00768     bool ok = false;
00769 
00770     if ((buf != NULL) && (pos < bufLen)) {
00771         uint16_t contentLen = 0U;
00772         pos += 1U; /* skip tag byte */
00773         if (derReadLength(buf, bufLen, pos, contentLen)) {
00774             if ((pos + contentLen) <= bufLen) {
00775                 pos += contentLen;
00776                 ok   = true;
```

```
00777             }
00778         }
00779     }
00780
00781     return ok;
00782 }
00783
00784 /* Walk a DER X.509 Certificate to extract -- without any byte-pattern search:
00785  *   tbsMsgStart  offset of TBSCertificate SEQUENCE tag inside buf
00786  *   tbsMsgLen    total byte count of TBSCertificate (tag + length + content)
00787  *   pubKey65Ptr  pointer to 65-byte uncompressed EC point (0x04 || X || Y)
00788  *   sigPtr       pointer to the DER ECDSA signature bytes
00789  *   sigLen       byte count of the DER ECDSA signature
00790  * Returns false on any format or bounds error. */
00791 static bool derWalkMfCert(const uint8_t* buf, uint16_t bufLen,
00792                           uint16_t& tbsMsgStart, uint16_t& tbsMsgLen,
00793                           const uint8_t*& pubKey65Ptr,
00794                           const uint8_t*& sigPtr, uint8_t& sigLen) {
00795     bool     ok             = true;
00796     uint16_t pos            = 0U;
00797     uint16_t certContentLen = 0U;
00798     uint16_t tbsContentLen  = 0U;
00799     uint16_t tbsEnd         = 0U;
00800     uint16_t spkiContentLen = 0U;
00801     uint16_t bsLen          = 0U;
00802
00803     tbsMsgStart = 0U;
00804     tbsMsgLen   = 0U;
00805     pubKey65Ptr = NULL;
00806     sigPtr      = NULL;
00807     sigLen      = 0U;
00808
00809     if ((buf == NULL) || (bufLen == 0U)) {
00810         ok = false;
00811     }
00812
00813     /* - outer Certificate SEQUENCE - */
00814     if (ok) {
00815         if (buf[pos] != DER_TAG_SEQUENCE) {
00816             ok = false;
00817         }
00818     }
00819     if (ok) {
00820         pos += 1U;
00821         if (!derReadLength(buf, bufLen, pos, certContentLen)) {
00822             ok = false;
00823         }
00824     }
00825     if (ok) {
00826         if ((pos + certContentLen) > bufLen) {
00827             ok = false;
00828         }
00829     }
00830
00831     /* - TBSCertificate SEQUENCE (first child) - */
00832     if (ok) {
00833         if (buf[pos] != DER_TAG_SEQUENCE) {
00834             ok = false;
00835         }
00836     }
00837     if (ok) {
00838         tbsMsgStart = pos;
00839         pos += 1U;
00840         if (!derReadLength(buf, bufLen, pos, tbsContentLen)) {
00841             ok = false;
00842         }
00843     }
00844     if (ok) {
00845         tbsMsgLen = (pos - tbsMsgStart) + tbsContentLen;
00846         tbsEnd    = pos + tbsContentLen;
00847         if (tbsEnd > bufLen) {
00848             ok = false;
00849         }
00850     }
00851
00852     /* - Walk TBSCertificate fields in order - */
00853
00854     /* [0] EXPLICIT version -- present in X.509 v3 */
00855     if (ok && (pos < tbsEnd)) {
00856         if (buf[pos] == DER_TAG_CTX0) {
00857             if (!derSkipField(buf, tbsEnd, pos)) {
00858                 ok = false;
00859             }
00860         }
00861     }
00862
00863     /* serialNumber INTEGER */
```

```
00864     if (ok) {
00865         if (!derSkipField(buf, tbsEnd, pos)) {
00866             ok = false;
00867         }
00868     }
00869
00870     /* signature AlgorithmIdentifier SEQUENCE */
00871     if (ok) {
00872         if (!derSkipField(buf, tbsEnd, pos)) {
00873             ok = false;
00874         }
00875     }
00876
00877     /* issuer Name SEQUENCE */
00878     if (ok) {
00879         if (!derSkipField(buf, tbsEnd, pos)) {
00880             ok = false;
00881         }
00882     }
00883
00884     /* validity SEQUENCE */
00885     if (ok) {
00886         if (!derSkipField(buf, tbsEnd, pos)) {
00887             ok = false;
00888         }
00889     }
00890
00891     /* subject Name SEQUENCE */
00892     if (ok) {
00893         if (!derSkipField(buf, tbsEnd, pos)) {
00894             ok = false;
00895         }
00896     }
00897
00898     /* - SubjectPublicKeyInfo SEQUENCE - */
00899     if (ok) {
00900         if (pos >= tbsEnd) {
00901             ok = false;
00902         }
00903     }
00904     if (ok) {
00905         if (buf[pos] != DER_TAG_SEQUENCE) {
00906             ok = false;
00907         }
00908     }
00909     if (ok) {
00910         pos += 1U;
00911         if (!derReadLength(buf, bufLen, pos, spkiContentLen)) {
00912             ok = false;
00913         }
00914     }
00915     if (ok) {
00916         if ((pos + spkiContentLen) > bufLen) {
00917             ok = false;
00918         }
00919     }
00920
00921     /* Skip AlgorithmIdentifier SEQUENCE inside SubjectPublicKeyInfo */
00922     if (ok) {
00923         if (!derSkipField(buf, bufLen, pos)) {
00924             ok = false;
00925         }
00926     }
00927
00928     /* subjectPublicKey BIT STRING */
00929     if (ok) {
00930         if (pos >= bufLen) {
00931             ok = false;
00932         }
00933     }
00934     if (ok) {
00935         if (buf[pos] != DER_TAG_BIT_STRING) {
00936             ok = false;
00937         }
00938     }
00939     if (ok) {
00940         pos += 1U;
00941         if (!derReadLength(buf, bufLen, pos, bsLen)) {
00942             ok = false;
00943         }
00944     }
00945     if (ok) {
00946         if ((pos + bsLen) > bufLen) {
00947             ok = false;
00948         }
00949     }
00950     if (ok) {
```

```
00951         if (bsLen < (1U + static_cast<uint16_t>(DER_EC_POINT_BYTES))) {
00952             ok = false; /* too short: unused-bits byte + 65-byte EC point */
00953         }
00954     }
00955     if (ok) {
00956         if (buf[pos] != DER_BIT_UNUSED_ZERO) {
00957             ok = false; /* unused bits must be 0 */
00958         } else if (buf[pos + 1U] != DER_EC_UNCOMPRESSED) {
00959             ok = false; /* must be an uncompressed EC point */
00960         } else {
00961             pubKey65Ptr = buf + pos + 1U; /* points to: 0x04 || X[32] || Y[32] */
00962         }
00963     }
00964
00965     /* Jump to end of TBSCertificate, then skip signatureAlgorithm SEQUENCE */
00966     if (ok) {
00967         pos = tbsEnd;
00968         if (!derSkipField(buf, bufLen, pos)) {
00969             ok = false;
00970         }
00971     }
00972
00973     /* - signatureValue BIT STRING (third child of Certificate) - */
00974     if (ok) {
00975         if (pos >= bufLen) {
00976             ok = false;
00977         }
00978     }
00979     if (ok) {
00980         if (buf[pos] != DER_TAG_BIT_STRING) {
00981             ok = false;
00982         }
00983     }
00984     if (ok) {
00985         pos += 1U;
00986         if (!derReadLength(buf, bufLen, pos, bsLen)) {
00987             ok = false;
00988         }
00989     }
00990     if (ok) {
00991         if ((pos + bsLen) > bufLen) {
00992             ok = false;
00993         }
00994     }
00995     if (ok) {
00996         if (bsLen < 2U) {
00997             ok = false; /* need unused-bits byte + at least 1 signature byte */
00998         }
00999     }
01000     if (ok) {
01001         if (buf[pos] != DER_BIT_UNUSED_ZERO) {
01002             ok = false; /* unused bits must be 0 */
01003         }
01004     }
01005     if (ok) {
01006         uint16_t rawSigLen = bsLen - 1U;
01007         if (rawSigLen > 255U) {
01008             ok = false;
01009         } else {
01010             sigPtr = buf + pos + 1U; /* DER ECDSA signature, after unused-bits byte */
01011             sigLen = static_cast<uint8_t>(rawSigLen);
01012         }
01013     }
01014
01015     return ok;
01016 }
01017
01018 bool CW_SecureChannel::parseDerSigToRaw(const uint8_t* der, uint8_t derLen,
01019                                         uint8_t* raw64) {
01020     bool ret = false;
01021
01022     if ((der != NULL) && (raw64 != NULL) && (derLen >= 6U) && (der[0] == 0x30U)) {
01023         /* Validate outer SEQUENCE length against actual buffer (HIGH-04). */
01024         if ((uint8_t)(der[1] + 2U) > derLen) {
01025             return false;
01026         }
01027
01028         uint8_t pos = 2U;  /* skip SEQUENCE tag + length */
01029
01030         if (der[pos] == 0x02U) {
01031             pos++;
01032             uint8_t rLen = der[pos];
01033             pos++;
01034             /* Reject malformed r -- DER r is at most 33 bytes (32 + 1 zero pad) (HIGH-04). */
01035             if (rLen > 33U) {
01036                 return false;
01037             }
```

```
01038             if ((pos + rLen) <= derLen) {
01039                 const uint8_t* rPtr = der + pos;
01040                 pos += rLen;
01041 
01042                 if ((pos < derLen) && (der[pos] == 0x02U)) {
01043                     pos++;
01044                     uint8_t sLen = der[pos];
01045                     pos++;
01046                     /* Reject malformed s (HIGH-04). */
01047                     if (sLen > 33U) {
01048                         return false;
01049                     }
01050                     if ((pos + sLen) <= derLen) {
01051                         /* M-06: verify sum of both INTEGER fields matches the outer SEQUENCE length.
01052                          * Trailing garbage after s would indicate malformed/crafted DER. */
01053                         if ((pos + sLen) != (uint8_t)(2U + der[1])) {
01054                             return false;
01055                         }
01056                         const uint8_t* sPtr = der + pos;
01057 
01058                         memset(raw64, 0U, 64U);
01059 
01060                         /* r: strip optional leading 0x00 padding byte */
01061                         if ((rLen == 33U) && (rPtr[0] == 0x00U)) { rPtr++; rLen = 32U; }
01062                         if (rLen <= 32U) {
01063                             (void)CW_Utils::safe_memcpy(raw64 + (32U - rLen), static_cast<size_t>(32U
      + rLen), rPtr, rLen);
01064                         }
01065 
01066                         /* s: strip optional leading 0x00 padding byte */
01067                         if ((sLen == 33U) && (sPtr[0] == 0x00U)) { sPtr++; sLen = 32U; }
01068                         if (sLen <= 32U) {
01069                             (void)CW_Utils::safe_memcpy(raw64 + 32U + (32U - sLen),
      static_cast<size_t>(sLen), sPtr, sLen);
01070                         }
01071 
01072                         ret = true;
01073                     }
01074                 }
01075             }
01076         }
01077     }
01078 
01079     return ret;
01080 }
01081 
01082 bool CW_SecureChannel::verifyEcdsaSha256(const uint8_t* pubKey64,
01083                                          const uint8_t* message, uint16_t msgLen,
01084                                          const uint8_t* derSig, uint8_t derSigLen) {
01085     bool result = false;
01086     uint8_t hash[32U] = { 0U };
01087     uint8_t rawSig[64U] = { 0U };
01088 
01089     bool hashOk = _crypto.sha256(message, msgLen, hash);
01090 
01091     if (hashOk && parseDerSigToRaw(derSig, derSigLen, rawSig)) {
01092         result = _crypto.ecdsaVerify(pubKey64, hash, sizeof(hash), rawSig, CW_CURVE_SECP256R1);
01093     }
01094 
01095     return result;
01096 }
01097 
01098 bool CW_SecureChannel::getManufacturerCertificate(uint8_t* cert, uint16_t& certLen) {
01099     bool ret = false;
01100     certLen = 0U;
01101 
01102     if (cert != NULL) {
01103         const uint8_t APDU_P2_IDX = 3U;  /* P2 field offset in ISO 7816-4 APDU header */
01104         uint8_t apdu[5U] = { 0x80U, 0xF7U, 0x00U, 0x00U, 0x00U };
01105         uint8_t response[RESPONSE_GETMANUFACTURERCERT_PAGE_IN_BYTES];
01106         uint16_t responseLen = static_cast<uint16_t>(sizeof(response));
01107 
01108         if (!_driver.sendAPDULarge(apdu, static_cast<uint8_t>(sizeof(apdu)), response,
01109                                    responseLen)) {
01110 #if CW_DEBUG_LOGGING
01111             _logger.println(F("getManufacturerCertificate APDU failed."));
01112 #endif
01113             return false;
01114         }
01115         if (responseLen > static_cast<uint16_t>(sizeof(response))) {
01116 #if CW_DEBUG_LOGGING
01117             _logger.println(F("getManufacturerCertificate: driver reported overflow."));
01118 #endif
01119             return false;
01120         }
01121 
01122         if (checkStatusWord(response, responseLen, 0x90U, 0x00U)) {
```

```
01123
01124             uint16_t dataBytes = static_cast<uint16_t>(
01125                 responseLen - static_cast<uint16_t>(RESPONSE_STATUS_WORDS_IN_BYTES));
01126
01127             if (dataBytes >= 2U) {
01128                 uint16_t totalCertLen = (static_cast<uint16_t>(response[0]) << 8U)
01129                                         | static_cast<uint16_t>(response[1]);
01130
01131                 if (totalCertLen <= CW_MANUF_CERT_MAX_BYTES) {
01132                     uint16_t certInPage = static_cast<uint16_t>(dataBytes - 2U);
01133                     if (certInPage > totalCertLen) {
01134                         certInPage = totalCertLen;
01135                     }
01136                     (void)CW_Utils::safe_memcpy(cert, CW_MANUF_CERT_MAX_BYTES, response + 2U,
01137                                                 static_cast<size_t>(certInPage));
01138                     certLen = certInPage;
01139
01140                     uint8_t pageIdx = 1U;
01141                     while ((certLen < totalCertLen) && (pageIdx < 8U)) {
01142                         apdu[APDU_P2_IDX] = pageIdx;
01143                         responseLen = static_cast<uint16_t>(sizeof(response));
01144
01145                         if (!_driver.sendAPDULarge(apdu, static_cast<uint8_t>(sizeof(apdu)),
01146                                                    response, responseLen)) {
01147                             break;
01148                         }
01149                         if (responseLen > static_cast<uint16_t>(sizeof(response))) {
01150 #if CW_DEBUG_LOGGING
01151                             _logger.println(F("getManufacturerCertificate: driver reported
    overflow."));
01152 #endif
01153                             return false;
01154                         }
01155                         if (!checkStatusWord(response, responseLen, 0x90U, 0x00U)) {
01156                             break;
01157                         }
01158
01159                         uint16_t pageData = static_cast<uint16_t>(
01160                             responseLen - static_cast<uint16_t>(RESPONSE_STATUS_WORDS_IN_BYTES));
01161                         uint16_t remaining = static_cast<uint16_t>(totalCertLen - certLen);
01162                         if (pageData > remaining) {
01163                             pageData = remaining;
01164                         }
01165
01166                         if ((certLen + pageData) > static_cast<uint16_t>(CW_MANUF_CERT_MAX_BYTES)) {
01167                             break;
01168                         }
01169                         (void)CW_Utils::safe_memcpy(cert + certLen,
01170                                                     CW_MANUF_CERT_MAX_BYTES -
    static_cast<size_t>(certLen),
01171                                                     response, static_cast<size_t>(pageData));
01172                         certLen = static_cast<uint16_t>(certLen + pageData);
01173                         pageIdx++;
01174                     }
01175
01176                     ret = (certLen == totalCertLen);
01177                     if (!ret) {
01178 #if CW_DEBUG_LOGGING
01179                         _logger.println(F("getManufacturerCertificate: incomplete."));
01180 #endif
01181                     }
01182                 } else {
01183 #if CW_DEBUG_LOGGING
01184                     _logger.println(F("getManufacturerCertificate: cert too large."));
01185 #endif
01186                 }
01187             }
01188         }
01189     }
01190
01191     return ret;
01192 }
01193
01194 bool CW_SecureChannel::preFetchManufacturerCert() {
01195     _cachedMfCertLen = 0U;
01196     bool result = getManufacturerCertificate(s_mfCertBuf, _cachedMfCertLen);
01197     if (!result) {
01198         _cachedMfCertLen = 0U;
01199     }
01200     return result;
01201 }
01202
01227 uint8_t CW_SecureChannel::verifyCertificateChain(const uint8_t* cardCert,
01228                                                  uint8_t cardCertLen) {
01229     uint8_t result = CW_CERT_OK;
01230
01231     if ((cardCert == NULL) || (cardCertLen < 80U) || (cardCert[0] != 0x43U)) {
```

```
01232 #if CW_DEBUG_LOGGING
01233         _logger.println(F("verifyCert: invalid card cert."));
01234 #endif
01235         result = CW_CERT_FORMAT_ERROR;
01236     }
01237
01238     /* Consume the pre-fetched cert (filled by preFetchManufacturerCert() before
01239      * getCardCertificate() was called).  Fall back to fetching now if none cached --
01240      * this will fail on cards whose state machine has already advanced past INS=F7. */
01241     uint16_t mfCertLen = _cachedMfCertLen;
01242     _cachedMfCertLen = 0U;
01243
01244     if (result == CW_CERT_OK) {
01245         if (mfCertLen == 0U) {
01246             if (!getManufacturerCertificate(s_mfCertBuf, mfCertLen) || (mfCertLen < 20U)) {
01247 #if CW_DEBUG_LOGGING
01248                 _logger.println(F("verifyCert: failed to get mfr cert."));
01249 #endif
01250                 result = CW_CERT_FORMAT_ERROR;
01251             }
01252         } else if (mfCertLen < 20U) {
01253 #if CW_DEBUG_LOGGING
01254             _logger.println(F("verifyCert: pre-fetched mfr cert too short."));
01255 #endif
01256             result = CW_CERT_FORMAT_ERROR;
01257         } else {
01258             /* Pre-fetched cert in s_mfCertBuf is valid -- no APDU needed. */
01259         }
01260     }
01261
01262     /* MED-05: replace byte-pattern search with a structural DER walker that
01263      * enforces field order.  This prevents attacker-planted OID sequences in
01264      * unsigned extensions from being picked up as the device public key. */
01265     uint16_t       tbsMsgStart = 0U;
01266     uint16_t       tbsMsgLen   = 0U;
01267     const uint8_t* pubKey65Ptr = NULL;
01268     const uint8_t* mfSigPtr    = NULL;
01269     uint8_t        mfSigLen    = 0U;
01270
01271     if (result == CW_CERT_OK) {
01272         if (!derWalkMfCert(s_mfCertBuf, mfCertLen,
01273                            tbsMsgStart, tbsMsgLen,
01274                            pubKey65Ptr,
01275                            mfSigPtr, mfSigLen)) {
01276 #if CW_DEBUG_LOGGING
01277             _logger.println(F("verifyCert: DER walk of mfr cert failed."));
01278 #endif
01279             result = CW_CERT_KEY_NOT_FOUND;
01280         }
01281     }
01282
01283     const uint8_t* devicePubKey64 = NULL;
01284     if (result == CW_CERT_OK) {
01285         devicePubKey64 = pubKey65Ptr + 1U; /* skip the 0x04 uncompressed prefix */
01286
01287         const uint8_t* mfMsg    = s_mfCertBuf + tbsMsgStart;
01288         uint16_t       mfMsgLen = tbsMsgLen;
01289
01290         bool mfVerified = false;
01291         for (uint8_t i = 0U; i < CW_TRUSTED_CA_COUNT; i++) {
01292             if (verifyEcdsaSha256(CW_TRUSTED_CA_KEYS[i],
01293                                   mfMsg, mfMsgLen,
01294                                   mfSigPtr, mfSigLen)) {
01295                 mfVerified = true;
01296                 break;
01297             }
01298         }
01299         if (!mfVerified) {
01300 #if CW_DEBUG_LOGGING
01301             _logger.println(F("verifyCert: mfr cert sig INVALID -- card NOT genuine."));
01302 #endif
01303             result = CW_CERT_MANUF_SIG_INVALID;
01304         }
01305 #if CW_DEBUG_LOGGING
01306         else {
01307             _logger.println(F("Manufacturer cert signature OK."));
01308         }
01309 #endif
01310     }
01311
01312     const uint8_t CARD_CERT_MSG_LEN = 74U;
01313     if (result == CW_CERT_OK) {
01314         if (cardCertLen <= CARD_CERT_MSG_LEN) {
01315 #if CW_DEBUG_LOGGING
01316             _logger.println(F("verifyCert: card cert too short for sig."));
01317 #endif
01318             result = CW_CERT_FORMAT_ERROR;
```

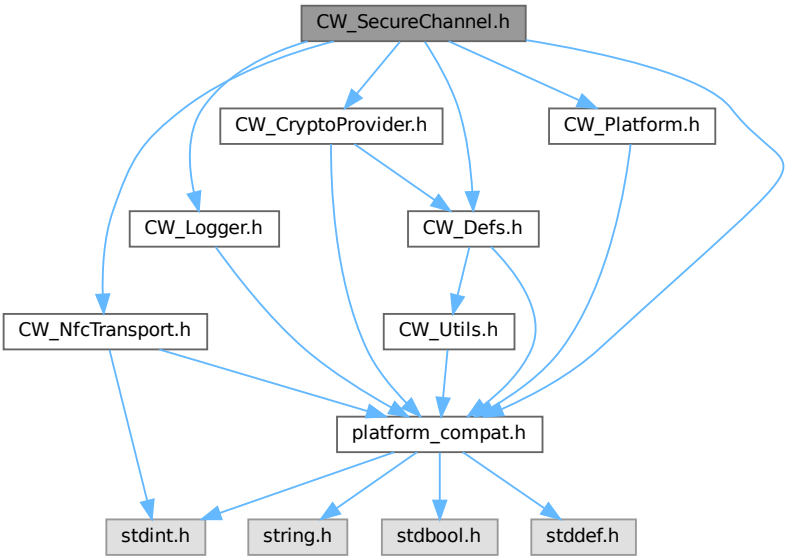
```
01319         }
01320         else {
01321             const uint8_t* cardSig    = cardCert + CARD_CERT_MSG_LEN;
01322             uint8_t        cardSigLen = cardCertLen - CARD_CERT_MSG_LEN;
01323 
01324             if (!verifyEcdsaSha256(devicePubKey64,
01325                                    cardCert, CARD_CERT_MSG_LEN,
01326                                    cardSig, cardSigLen)) {
01327 #if CW_DEBUG_LOGGING
01328                 _logger.println(F("verifyCert: card cert sig INVALID."));
01329 #endif
01330                 result = CW_CERT_CARD_SIG_INVALID;
01331             }
01332 #if CW_DEBUG_LOGGING
01333             else {
01334                 _logger.println(F("Card cert signature OK."));
01335             }
01336 #endif
01337         }
01338     }
01339 
01340     if (result == CW_CERT_OK) {
01341         if ((cardCertLen <= (1U + CW_CERT_NONCE_SIZE)) ||
01342             (!CW_Utils::secure_compare(cardCert + 1U, _lastNonce, CW_CERT_NONCE_SIZE))) { /* M-03:
      constant-time nonce compare */
01343 #if CW_DEBUG_LOGGING
01344             _logger.println(F("verifyCert: nonce mismatch -- possible replay."));
01345 #endif
01346             result = CW_CERT_NONCE_MISMATCH;
01347         }
01348     }
01349 
01350 #if CW_DEBUG_LOGGING
01351     if (result == CW_CERT_OK) {
01352         _logger.println(F("Certificate chain OK. Card is genuine."));
01353     }
01354 #endif
01355 
01356     /* Wipe manufacturer cert buffer -- it held the card device public key (M-01). */
01357     CW_Utils::secure_wipe(s_mfCertBuf, sizeof(s_mfCertBuf));
01358 
01359     return result;
01360 }
```

4.17 CW_SecureChannel.h File Reference

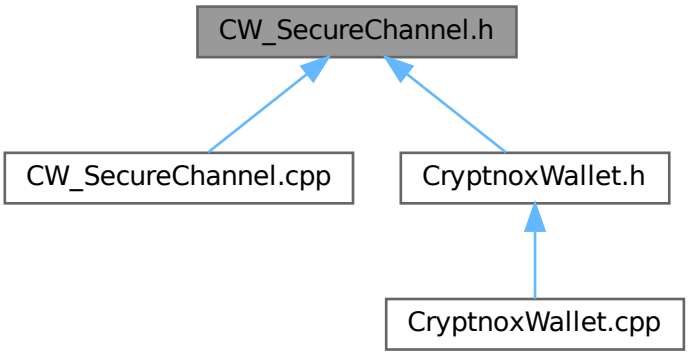
Cryptnox secure channel protocol over NFC.

```
#include "platform_compat.h"
#include "CW_NfcTransport.h"
#include "CW_Logger.h"
#include "CW_CryptoProvider.h"
#include "CW_Platform.h"
#include "CW_Defs.h"
```

Include dependency graph for CW_SecureChannel.h:



This graph shows which files directly or indirectly include this file:



Classes

- class CW_SecureChannel

  Implements the Cryptnox secure channel protocol over NFC.

Macros

- #define CW_PAIRING_DATA "Cryptnox Basic CommonPairingData"
- #define CW_PAIRING_DATA_BYTES (sizeof(CW_PAIRING_DATA) - 1U)

4.17.1 Detailed Description

Cryptnox secure channel protocol over NFC.
Declares CW_SecureChannel, responsible for every low-level APDU exchange required to establish and use an authenticated, encrypted session with a Cryptnox Hardware Wallet:

- SELECT AID
- Manufacturer + card certificate retrieval and chain verification
- Ephemeral key extraction and ECDH session key derivation
- MUTUALLY AUTHENTICATE
- AES-CBC + MAC secure messaging (encrypt / decrypt / SW check)

The class is composed inside CryptnoxWallet and is not normally used directly by application code.
Definition in file CW_SecureChannel.h.

4.17.2 Macro Definition Documentation

4.17.2.1 CW_PAIRING_DATA

```
#define CW_PAIRING_DATA "Cryptnox Basic CommonPairingData"
```

Definition at line 30 of file CW_SecureChannel.h.

4.17.2.2 CW_PAIRING_DATA_BYTES

```
#define CW_PAIRING_DATA_BYTES (sizeof(CW_PAIRING_DATA) - 1U)
```

Definition at line 31 of file CW_SecureChannel.h.

4.18 CW_SecureChannel.h

Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00022
00023 #ifndef CW_SECURECHANNEL_H
00024 #define CW_SECURECHANNEL_H
00025
00026 /**********************************************************************
00027  * 1. Public constants
00028  **********************************************************************/
00029
00030 #define CW_PAIRING_DATA       "Cryptnox Basic CommonPairingData"
00031 #define CW_PAIRING_DATA_BYTES (sizeof(CW_PAIRING_DATA) - 1U)
00032
00033 /**********************************************************************
00034  * 2. Included files
00035  **********************************************************************/
00036
00037 #include "platform_compat.h"
00038 #include "CW_NfcTransport.h"
00039 #include "CW_Logger.h"
00040 #include "CW_CryptoProvider.h"
00041 #include "CW_Platform.h"
00042 #include "CW_Defs.h"          /* for CW_SecureSession, CW_Curve, and constants */
00043
00044 /**********************************************************************
00045  * 2. Class declaration
00046  **********************************************************************/
00047
00064 class CW_SecureChannel {
00065 public:
00074     CW_SecureChannel(CW_NfcTransport& driver, CW_Logger& logger,
00075                      CW_CryptoProvider& crypto, CW_Platform& platform);
00076
00077     CW_SecureChannel(const CW_SecureChannel&) = delete;
00078     CW_SecureChannel& operator=(const CW_SecureChannel&) = delete;
00079
```

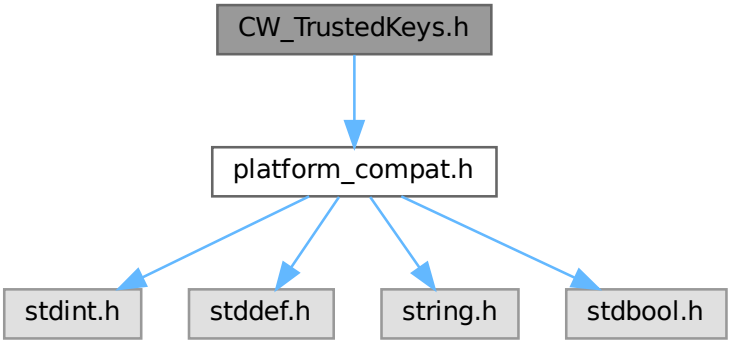
```
00084     bool begin();
00085
00090     bool inListPassiveTarget();
00091
00095     void resetReader();
00096
00101     bool printFirmwareVersion();
00102
00107     bool selectApdu();
00108
00120     bool getCardCertificate(uint8_t* cardCertificate, uint8_t& cardCertificateLength);
00121
00130     bool extractCardEphemeralKey(const uint8_t* cardCertificate,
00131                                  uint8_t* cardEphemeralPubKey,
00132                                  uint8_t* fullEphemeralPubKey65 = NULL);
00133
00151     bool openSecureChannel(uint8_t* salt,
00152                            uint8_t* clientPublicKey,
00153                            uint8_t* clientPrivateKey,
00154                            CW_Curve sessionCurve);
00155
00183     bool mutuallyAuthenticate(CW_SecureSession& session,
00184                               const uint8_t* salt,
00185                               uint8_t* clientPublicKey,
00186                               const uint8_t* clientPrivateKey,
00187                               CW_Curve sessionCurve,
00188                               const uint8_t* cardEphemeralPubKey);
00189
00197     bool getManufacturerCertificate(uint8_t* cert, uint16_t& certLen);
00198
00209     bool preFetchManufacturerCert();
00210
00238     uint8_t verifyCertificateChain(const uint8_t* cardCert, uint8_t cardCertLen);
00239
00269     bool aesCbcEncrypt(CW_SecureSession& session,
00270                        const uint8_t apdu[], uint16_t apduLength,
00271                        const uint8_t data[], uint16_t dataLength,
00272                        uint8_t* decryptedOutput = NULL,
00273                        uint16_t* decryptedOutputLength = NULL);
00274
00294     bool aesCbcDecrypt(const CW_SecureSession& session,
00295                        uint8_t* response, size_t responseLen,
00296                        uint8_t* macValue,
00297                        uint8_t* decryptedOutput = NULL,
00298                        uint16_t* decryptedOutputLength = NULL);
00299
00309     bool checkStatusWord(const uint8_t* response, uint16_t responseLength,
00310                          uint8_t sw1Expected, uint8_t sw2Expected);
00311
00312 private:
00313     CW_NfcTransport&   _driver;
00314     CW_Logger&         _logger;
00315     CW_CryptoProvider& _crypto;
00316     CW_Platform&       _platform;
00317
00319     uint8_t _lastNonce[CW_CERT_NONCE_SIZE];
00320
00322     uint16_t _cachedMfCertLen;
00323
00324     static bool parseDerSigToRaw(const uint8_t* der, uint8_t derLen,
00325                                  uint8_t* raw64);
00326
00327     bool verifyEcdsaSha256(const uint8_t* pubKey64,
00328                            const uint8_t* message, uint16_t msgLen,
00329                            const uint8_t* derSig, uint8_t derSigLen);
00330
00331 #ifdef CW_FUZZ_BUILD
00332     friend struct DerFuzzTarget;
00333 #endif
00334 };
00335
00336 #endif // CW_SECURECHANNEL_H
```

4.19 CW_TrustedKeys.h File Reference

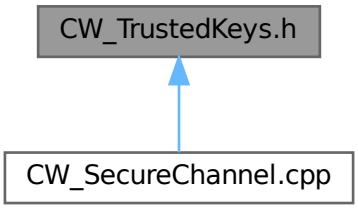
Cryptnox CA public keys used for offline certificate verification.

`#include "platform_compat.h"`

Include dependency graph for CW_TrustedKeys.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define CW_TRUSTED_CA_COUNT (1U)

Variables

- static const uint8_t CW_CA_DLT_PUBKEY [64]
- static const uint8_t ∗const CW_TRUSTED_CA_KEYS [CW_TRUSTED_CA_COUNT]

4.19.1 Detailed Description

Cryptnox CA public keys used for offline certificate verification.
Holds the trusted secp256r1 public keys that CW_SecureChannel::verifyCertificateChain checks the manufacturer certificate against. Each key is stored as 64 raw bytes (X[32] || Y[32], no 0x04 prefix) so it can be passed directly to `uECC_verify()`.
Source: https://verify.cryptnox.tech/certificates/
To update a key: download the .crt file from the URL above and extract the EC public-key coordinates with:

```
python -c "from cryptography import x509; \
  c=x509.load_pem_x509_certificate(open('CERT.crt','rb').read()); \
  n=c.public_key().public_numbers(); \
```

```
  print(hex(n.x)); print(hex(n.y))"
```
Definition in file CW_TrustedKeys.h.

4.19.2 Macro Definition Documentation

4.19.2.1 CW_TRUSTED_CA_COUNT

```
#define CW_TRUSTED_CA_COUNT (1U)
```
Definition at line 46 of file CW_TrustedKeys.h.
Referenced by CW_SecureChannel::verifyCertificateChain().

4.19.3 Variable Documentation

4.19.3.1 CW_CA_DLT_PUBKEY

```
const uint8_t CW_CA_DLT_PUBKEY[64]  [static]
```
Initial value:
```
= {

    0x64, 0x7a, 0x11, 0x2c, 0x2a, 0xcf, 0x5a, 0x49,
    0xb5, 0x2b, 0x32, 0x1c, 0xbd, 0xcf, 0x5a, 0x9e,
    0xa3, 0x07, 0xfc, 0x4c, 0xcd, 0x33, 0xaa, 0x78,
    0xb9, 0xb8, 0x05, 0x5d, 0xd8, 0x4d, 0x03, 0xae,

    0xda, 0xb0, 0x2c, 0xc7, 0x20, 0xa7, 0x80, 0xcb,
    0x1f, 0xf2, 0x80, 0xaf, 0x50, 0x77, 0x4a, 0x6c,
    0xdc, 0x15, 0x7e, 0xfa, 0x23, 0x5e, 0xa2, 0x53,
    0x11, 0xa4, 0x2b, 0x4c, 0xf5, 0x7d, 0x88, 0x61
}
```
Definition at line 32 of file CW_TrustedKeys.h.

4.19.3.2 CW_TRUSTED_CA_KEYS

```
const uint8_t* const CW_TRUSTED_CA_KEYS[CW_TRUSTED_CA_COUNT]  [static]
```
Initial value:
```
= {
    CW_CA_DLT_PUBKEY
}
```
Definition at line 50 of file CW_TrustedKeys.h.
Referenced by CW_SecureChannel::verifyCertificateChain().

4.20 CW_TrustedKeys.h

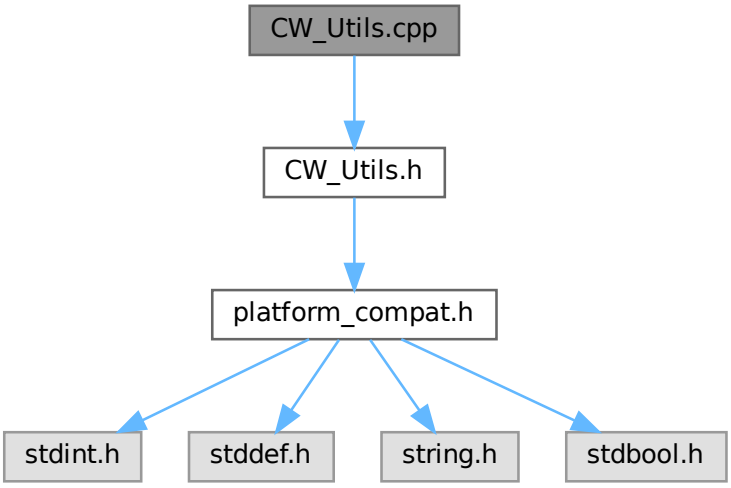
Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00026
00027 #ifndef CW_TRUSTEDKEYS_H
00028 #define CW_TRUSTEDKEYS_H
00029
00030 #include "platform_compat.h"
00031
00032 /* CRYPTNOX_DLT_CARDS_CA -- signs manufacturer certificates for DLT cards.
00033  * Curve: secp256r1. Issued by CRYPTNOX INTERMEDIATE CA #2. */
00034 static const uint8_t CW_CA_DLT_PUBKEY[64] = {
00035     /* X */
00036     0x64, 0x7a, 0x11, 0x2c, 0x2a, 0xcf, 0x5a, 0x49,
00037     0xb5, 0x2b, 0x32, 0x1c, 0xbd, 0xcf, 0x5a, 0x9e,
00038     0xa3, 0x07, 0xfc, 0x4c, 0xcd, 0x33, 0xaa, 0x78,
00039     0xb9, 0xb8, 0x05, 0x5d, 0xd8, 0x4d, 0x03, 0xae,
00040     /* Y */
00041     0xda, 0xb0, 0x2c, 0xc7, 0x20, 0xa7, 0x80, 0xcb,
00042     0x1f, 0xf2, 0x80, 0xaf, 0x50, 0x77, 0x4a, 0x6c,
00043     0xdc, 0x15, 0x7e, 0xfa, 0x23, 0x5e, 0xa2, 0x53,
00044     0x11, 0xa4, 0x2b, 0x4c, 0xf5, 0x7d, 0x88, 0x61
00045 };
00046
```

```
00047 /* Number of trusted CA keys in the table below. */
00048 #define CW_TRUSTED_CA_COUNT (1U)
00049
00050 /* Table of all trusted CA public keys (each 64 bytes, X||Y).
00051  * verifyCertificateChain() tries every entry until one succeeds. */
00052 static const uint8_t* const CW_TRUSTED_CA_KEYS[CW_TRUSTED_CA_COUNT] = {
00053     CW_CA_DLT_PUBKEY
00054 };
00055
00056 #endif /* CW_TRUSTEDKEYS_H */
```

4.21 CW_Utils.cpp File Reference

Implementation of the platform-independent utility helpers.
`#include "CW_Utils.h"`
Include dependency graph for CW_Utils.cpp:



4.21.1 Detailed Description

Implementation of the platform-independent utility helpers.
CW_Utils::secure_compare iterates over the full length to avoid leaking byte-position information through timing. CW_Utils::secure_wipe uses a volatile pointer so the writes cannot be optimised away. CW_Utils::safe_memcpy validates pointers, length, and source/destination overlap before delegating to memcpy.
Definition in file CW_Utils.cpp.

4.22 CW_Utils.cpp

Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00016
00017 #include "CW_Utils.h"
00018
00022 bool CW_Utils::secure_compare(const uint8_t* a, const uint8_t* b, size_t len) {
00023     bool ret = false;
```

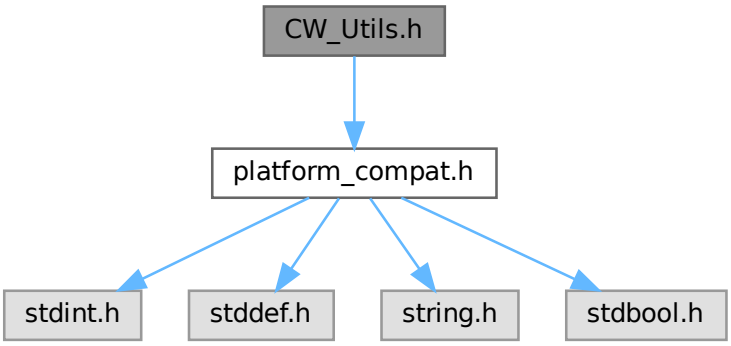
```
00024     if ((a != NULL) && (b != NULL) && (len > 0U)) {
00025         uint8_t diff = 0U;
00026         for (size_t i = 0U; i < len; i++) {
00027             diff |= a[i] ^ b[i];
00028         }
00029         ret = (diff == 0U);
00030     }
00031     return ret;
00032 }
00033
00037 void CW_Utils::secure_wipe(uint8_t* buf, size_t len) {
00038     if ((buf != NULL) && (len > 0U)) {
00039         volatile uint8_t* p = buf;
00040         for (size_t i = 0U; i < len; i++) {
00041             p[i] = 0U;
00042         }
00043     }
00044 }
00045
00050 bool CW_Utils::safe_memcpy(uint8_t* dst, size_t dstSize,
00051                                 const uint8_t* src, size_t count) {
00052     bool ret = false;
00053     if ((dst != NULL) && (src != NULL) && (count > 0U) && (count <= dstSize)) {
00054         bool overlap = (dst < (src + count)) && (src < (dst + dstSize));
00055         if (!overlap) {
00056             memcpy(dst, src, count);
00057             ret = true;
00058         }
00059     }
00060     return ret;
00061 }
```

4.23 CW_Utils.h File Reference

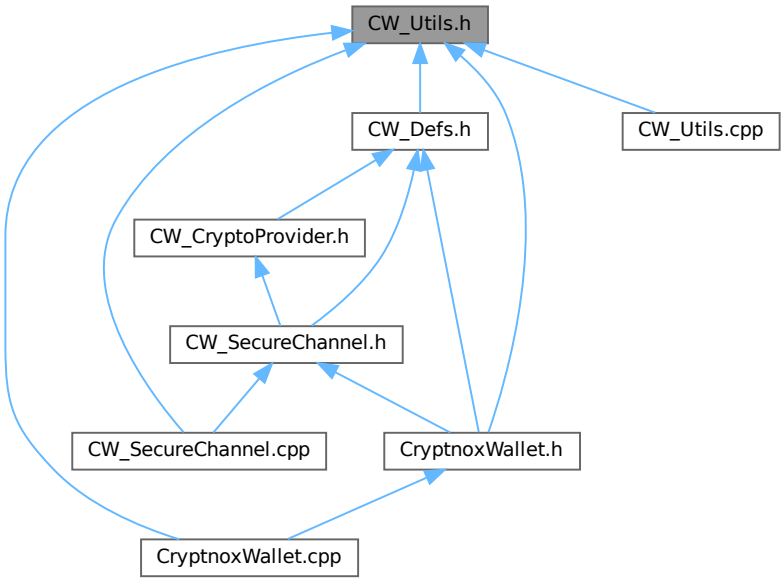
Platform-independent security and memory utilities.

`#include "platform_compat.h"`

Include dependency graph for CW_Utils.h:



This graph shows which files directly or indirectly include this file:



Classes

- class CW_Utils

  Portable utility functions for cryptographic and security operations.

4.23.1 Detailed Description

Platform-independent security and memory utilities.
Declares CW_Utils, a small collection of helpers used throughout the SDK that have no platform dependencies:

- constant-time buffer comparison (timing-side-channel safe)
- guaranteed-not-elided secure wipe of sensitive buffers
- bounds-checked, overlap-checked memcpy
- secure random byte generation (delegated to a platform impl)

Hardware-specific helpers (e.g. TRNG bring-up) live in the concrete crypto provider rather than here.
Definition in file CW_Utils.h.

4.24 CW_Utils.h

Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00020
00021 #ifndef CW_UTILS_H
00022 #define CW_UTILS_H
00023
00024 /**************************************************************************
```

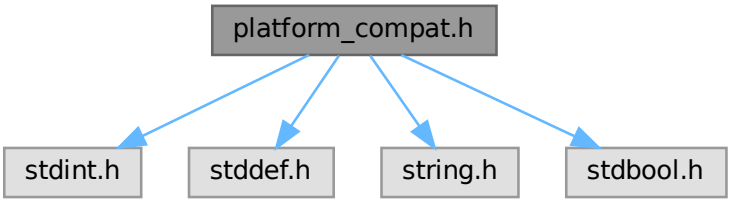
```
00025  * 1. Included files
00026  ***************************************************************************/
00027
00028 #include "platform_compat.h"
00029
00030 /***************************************************************************
00031  * 2. Class declaration
00032  ***************************************************************************/
00033
00045 class CW_Utils {
00046 public:
00059     static bool secure_compare(const uint8_t* a, const uint8_t* b, size_t len);
00060
00070     static void secure_wipe(uint8_t* buf, size_t len);
00071
00081     static bool safe_memcpy(uint8_t* dst, size_t dstSize,
00082                             const uint8_t* src, size_t count);
00083
00095     static bool fill_secure_random(uint8_t* dest, size_t len);
00096 };
00097
00098 #endif // CW_UTILS_H
```

4.25 platform_compat.h File Reference

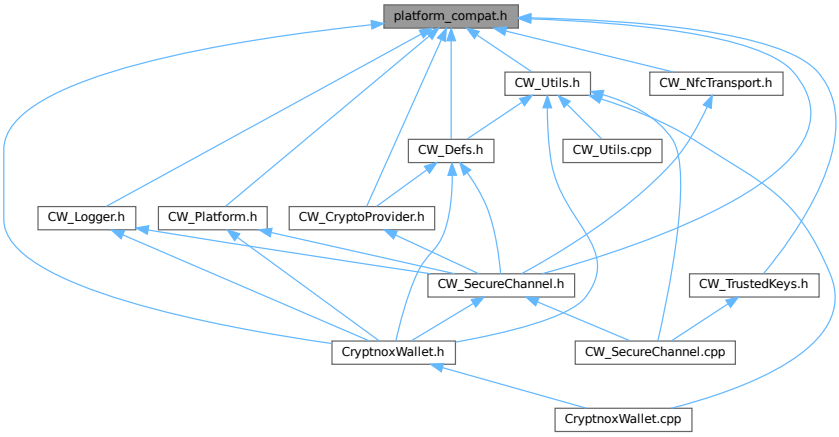
Arduino compatibility shims for non-Arduino (plain C++) builds.

```
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include <stdbool.h>
```

Include dependency graph for platform_compat.h:



This graph shows which files directly or indirectly include this file:



Classes

- class __FlashStringHelper

Macros

- #define DEC 10
- #define HEX 16
- #define OCT 8
- #define BIN 2
- #define F(string_literal)

4.25.1 Detailed Description

Arduino compatibility shims for non-Arduino (plain C++) builds.
Include this file instead of <Arduino.h> in platform-independent headers. When building for Arduino, ARDUINO is already defined by the toolchain and this file is a no-op — Arduino.h has already provided all these definitions. When building outside Arduino (desktop, ESP-IDF, CI) this file provides the minimum set of defines/types needed so the SDK headers compile cleanly.
Definition in file platform_compat.h.

4.25.2 Macro Definition Documentation

4.25.2.1 BIN

```
#define BIN 2
```

Definition at line 37 of file platform_compat.h.

4.25.2.2 DEC

```
#define DEC 10
```

Definition at line 34 of file platform_compat.h.
Referenced by CW_Logger::print(), CW_Logger::print(), CW_Logger::print(), CW_Logger::print(), CW_Logger::println(), CW_Logger::println(), CW_Logger::println(), and CW_Logger::println().

4.25.2.3 F

```
#define F(
              string_literal)
```

Value:

```
(string_literal)
```

Definition at line 42 of file platform_compat.h.
Referenced by CW_SecureChannel::aesCbcDecrypt(), CW_SecureChannel::aesCbcEncrypt(), CW_SecureChannel::checkStatusWord(), CryptnoxWallet::connect(), CryptnoxWallet::debugPrintSignature(), CryptnoxWallet::establishSecureChannel(), CryptnoxWallet::extractRawSignature(), CW_SecureChannel::getCardCertif CryptnoxWallet::getCardInfo(), CW_SecureChannel::getManufacturerCertificate(), CW_SecureChannel::mutuallyAuther CW_SecureChannel::openSecureChannel(), CW_SecureChannel::selectApdu(), CryptnoxWallet::sendSignApdu(), CryptnoxWallet::validateSignRequest(), CW_SecureChannel::verifyCertificateChain(), CryptnoxWallet::verifyPin(), and CryptnoxWallet::writeUserData().

4.25.2.4 HEX

```
#define HEX 16
```

Definition at line 35 of file platform_compat.h.
Referenced by CW_SecureChannel::aesCbcDecrypt(), CW_SecureChannel::aesCbcEncrypt(), CW_SecureChannel::checkStatusWord(), CryptnoxWallet::debugPrintSignature(), and CryptnoxWallet::establishSecure

4.25.2.5 OCT

```
#define OCT 8
```

Definition at line 36 of file platform_compat.h.

4.26 platform_compat.h

Go to the documentation of this file.

```
00001 /*
00002  * SPDX-License-Identifier: LGPL-3.0-or-later
00003  * Copyright (c) 2026 Cryptnox SA
00004  */
00005
00006 #ifndef PLATFORM_COMPAT_H
00007 #define PLATFORM_COMPAT_H
00008
00019
00020 #ifdef ARDUINO
00021 /* On Arduino, pull in Arduino.h so SDK headers that reference __FlashStringHelper,
00022  * DEC/HEX/OCT/BIN, F(), or delay() compile regardless of whether the including
00023  * translation unit already included <Arduino.h>. Arduino.h transitively provides
00024  * <stdint.h>, <stddef.h>, <string.h>, etc., so no need to include them here. */
00025 #  include <Arduino.h>
00026 #else
00027 /* Off Arduino, include the C standard headers explicitly so the SDK compiles
00028  * without depending on what the including TU may or may not have pulled in. */
00029 #  include <stdint.h>
00030 #  include <stddef.h>
00031 #  include <string.h>
00032 #  include <stdbool.h>
00033
00034 #  define DEC  10
00035 #  define HEX  16
00036 #  define OCT   8
00037 #  define BIN   2
00038
00039 /* F("...") on Arduino returns __FlashStringHelper* to keep string literals in
00040  * flash.  On non-Arduino it is the identity macro, resolving to const char*. */
00041 class __FlashStringHelper {};
00042 #  define F(string_literal) (string_literal)
00043
00044 #endif /* ARDUINO / !ARDUINO */
00045
00046 #endif /* PLATFORM_COMPAT_H */
```

4.27 README.md File Reference