AMERICAN FORTRESS™

# WALLET SDK DOCUMENTATION

# WalletSDK Interface Specification (Beta V1.2)

This document defines the final version of the WalletSDK interface for integrating name-based contact resolution and signaling into crypto wallets. The SDK is designed to be **developer-friendly**, **reactive**, and **cryptographically secure**, while abstracting complex signaling logic. Consult the AmericanFortress™ whitepaper for an architectural overview of the system. This SDK is patent pending under US and foreign patents which are detailed in the AmericanFortress whitepaper.

## Overview

WalletSDK enables wallets to:

- Register human-readable names (sfnames) tied to BIP47 payment codes
- Add and manage contacts by sfname
- Resolve a contact's address on a specific chain
- Send signals (BIP-47 compatibility available)
- React to new contact additions via a callback
- Authenticate securely using private key signatures
- Derive payment code and pubkey from injected seed
- Retrieve wallet's receive addresses for specific counterparties and chains
- Check availability of an sfname before attempting registration

Once authenticated, the SDK operates under a single-user and single-name context for the duration of the session. Multiple name functionality will be available in a future version.

---

## Class: WalletSDK

## Constructor

### WalletSDK(std::string api_base_url);

Initializes the SDK with the API base URL.

---

## Authentication Methods

std::string request_challenge(Sfname user);

Requests a cryptographic challenge string for authentication.The wallet signs this challenge using its private key.

**Returns:** A challenge string to be signed.

---

bool authenticate(Sfname user, std::string signature);

Submits the signed challenge and, if valid, receives a JWT token.Establishes a session-bound user context for all subsequent SDK calls. This allows the SDK to communicate with the backend infrastructure.

**Returns:** true if authentication is successful.

---

## Identity Registration

bool is_name_available(Sfname user);

Checks if a given sfname is available for registration. In beta name registration is free. This is intended to be used unauthenticated — prior to the wallet user having a name.

**Returns:** true if available, false if already taken.

---

bool register_name(Sfname user, Pcode pcode);

Registers a new identity (sfname) with a payment code. This is intended to be used unauthenticated — prior to the wallet user having a name. This functionality is free in beta, in production the register_name method will provide payment hooks.

**Returns:** true if the name was successfully registered.

---

## Identity Derivation

void import_seed(std::vector<uint8_t> seed);

Imports a BIP39-derived seed generated externally. The SDK will internally derive the pubkey and BIP47 payment code.

---

Pcode get_my_payment_code();

Returns the BIP47 payment code derived from the last imported seed.

---

Pubkey get_my_pubkey();

Returns the public key derived from the imported seed. Optional for developers that need to display or verify it.

---

## Contact Management

bool add_contact(Contact contact);

Adds a contact entry to the current user's contact list. Used to prepare for sending and receiving funds to/from that contact.

**Returns:** true if contact was added successfully.

---

std::vector<Contact> retrieve_all_contacts();

Returns the full set of known contacts stored for this user.

---

void on_new_contact(std::function<void(Contact)> callback);

Registers a callback function that fires when a new contact is added —either via sync or SDK-internal logic. The callback is ideally the add_contact method for the wallet.

---

## Contact Resolution & Signaling

Address resolve_contact_address(Contact contact, std::string chain);

For sending funds, resolves the counterparty's receive address on the specified chain.
**Returns:** Address usable for blockchain transaction.

---

bool send_signal(Sfname to_contact, std::string chain);

Sends a signal to the contact for the specified chain. To insure funds delivery the wallet developer has to make sure that this **Returns:** true prior to sending the funds.

**Returns:** true if the signal was successfully sent.

---

std::vector<Address> get_my_receive_addresses(std::vector<Sfname> counterparties, std::vector<std::string> chain_codes);

Returns the addresses this wallet should add to it's internal monitor list to locate received funds from counterparties across the specified chains.

---

## Developer Usage Flow

1. Wallet imports a previously created seed. This has to be only performed once upon initialization of the SDK. Prior to production were extending this function such that

supports on disk encryption of the seed. We will additionally provide a variant of the SDK where the SDK doesn't require a seed import, but provides the required reference functions and test vectors for the SDK users (aka 3rd party wallet developers) to implement internally.

```
sdk.import_seed(seed);
auto pcode = sdk.get_my_payment_code();
```

2. Check if name is available and register. This is to initially register the name. In beta the names are free. In production payment mechanisms for the user to pay for a name will be included in the SDK.

```
if (sdk.is_name_available(Sfname{"alice"})) {
    sdk.register_name(Sfname{"alice"}, pcode);
}
```

3. Authenticate. We use the wallets own sfname to authenticate with our accelerators.

```
auto challenge = sdk.request_challenge(Sfname{"alice"});
auto signature = sign_with_private_key(challenge, privkey);
sdk.authenticate(Sfname{"alice"}, signature);
```

4. Add a contact. Done when a user sends money for the first time or manually adds a contact to the contact list.

```
Contact bob = { Sfname{"bob"}, bob_pcode, bob_pubkey };
sdk.add_contact(bob);
```

5. For sending funds the wallet can resolve counterparty contact's address via:

```
Address recipient = sdk.resolve_contact_address(bob, "ETH");
```

6. Prior to sending funds the counterparty must be signaled. If the signal was already sent, the SDK will not resend it. Each counterparty must be signaled once only, ever and the signal is permanent. Meaning, when the wallet is restored from seed it doesn't need to resend any signals.

```
SignalStatus = send_signal(bob, "ETH");
```

7. React to contact additions:

```
sdk.on_new_contact([](Contact c) {
    std::cout << "New contact added: " << c.name.value << std::endl;
});
```

8. Get all receive addresses for expected senders:

**auto** addresses = sdk.get_my_receive_addresses({ Sfname{"bob"} }, { "ETH", "BTC" });

---

## Notes

- Wallets may use their **own seed generation** methods and simply inject the derived seed using import_seed().
- The SDK will use the seed to derive pubkey and pcode internally.
- send_signal() must be called before funds are sent — this signals a new outgoing contact. Each contact needs to be notified only one time.
- get_contacts() provides recent/active entries. retrieve_all_contacts() returns the full list.

## Data Types

- struct Sfname
- **struct** Sfname {
  std::string value;
  };
- struct Pcode
- **struct** Pcode {
  std::string value;
  };
- struct Pubkey
- **struct** Pubkey {
  std::string hex;
  };
- struct Address
- **struct** Address {
  std::string value;
  std::string chain;
  };
- struct Contact
- **struct** Contact {
  Sfname name;
  Pcode pcode;

```
    Pubkey pubkey;
};
```