Specification

Infrastructure for Stateless Account Abstraction on Fuel

Date: February 2025

Version Information

Document Version:	1.0.0
Release Date:	February 2025
ZapWallet version tag:	v0.8.0
zap-contracts branch:	`dev_testing`
commit:	`135a6c628e458dbde1aa804dd746bb90e50a97df`
Compatible with:	• Sway v0.66.6 • Fuel-Core v0.40.0

Version History	
1.0.0	Initial specification release for tag v0.8.0

Notes:

- This specification describes the ZapWallet implementation as of February 2025
- All version numbers follow semantic versioning (MAJOR.MINOR.PATCH)

1 Introduction

The following specification is broken down in to the core parts that describe the Zap wallet architecture.

Predicate Wallet:
The predicate wallet (ZapWallet) is a stateless account abstraction wallet written in Sway for use on the Fuel network. The "wallet" is built from multiple predicates called "Modules" and a "Master" predicate. Both the Master and Modules contain the logic to validate specific transaction types signed by the owner of the wallet. The "Master" predicate is the asset holding address and serves as the central point of validation control for ZapWallet transactions.

Modules:
Modules in the ZapWallet architecture serve as specialized validation components that enable specific functionality while maintaining the wallet's security. Each module validates specific types of transactions and have strict input, output and validation criteria. A Module consists of the predicate code, a unique AssetId and a unique Address.

Manager:
The manager contract (ZapManager) serves as a coordination point in the ZapWallet architecture, managing critical aspects of wallet creation, operation and lifecycle. The ZapManager maintains three primary functions:

1. Wallet Initialization. The contract controls the initialization process for new ZapWallets by:

- Minting and distributing initial nonce native assets required for new ZapWallets using Modules 1-3
- Managing associations between Ethereum addresses and their corresponding-ZapWallet
- Ensuring a ZapWallet can not re-initialize.

2. Asset Management. As the central authority for asset creation, the contract:

- Acts as the sole minter for all module assets
- Controls nonce token creation and distribution
- Maintains integrity of module asset allocation

3. Upgrade. Controls the ability for version upgrades:

- Verifies wallet ownership through nonce asset validation
- Manages phased transition from initialization to upgrade states
- Tracks upgrade status through module asset verification
- Maintains version pair numbering for both V1 and V2 wallet implementations

1.1 ZapWallet Architecture

The ZapWallet architecture consists of a master predicate that coordinates with multiple specialized module predicates, as shown in Figure 1.
The Master predicate serves as the central authority for transactions and asset storage, while Modules provide specific functionality:

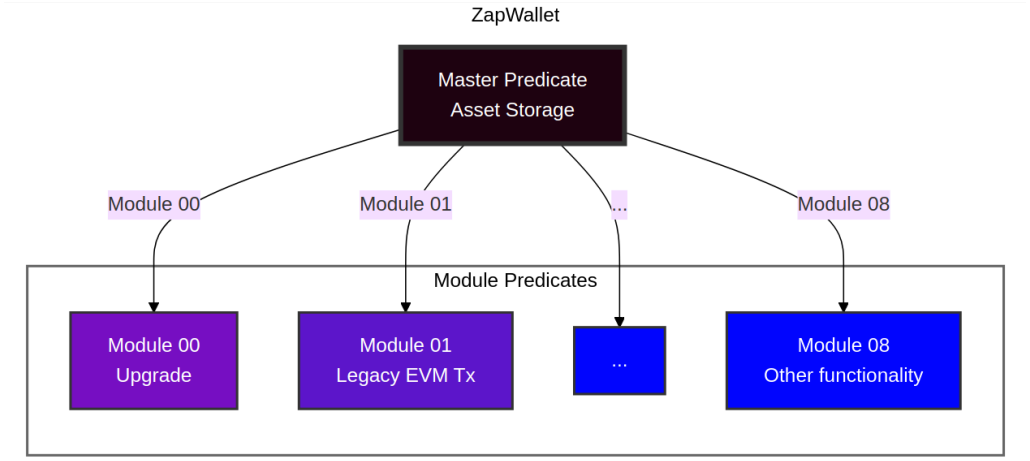

Figure 1: ZapWallet Architectural Overview

- **Module 00 (Upgrade):** Handles wallet upgrade operations
- **Module 01 (Legacy EVM Tx):** Processes legacy Ethereum transactions (first-price auction model)
- **Module 02 (EIP-1559 EVM Tx):** Processes Ethereum EIP-1559 transactions (base fee + priority fee model) for Fuel BASE_ASSET only
- **Module 03 (EIP-1559 EVM Tx):** Processes ERC20 style Ethereum EIP-1559 transactions (base fee + priority fee model) for Fuel native assets; SRC20 etc
- **Module 04 (TXID Witnessing):** Processes any type of Fuel transaction with the owner witnessing the transaction ID
- **Module 05 (Native Transfer):** Processes any Fuel native asset transaction (with the ability to have gas sponsorship) validated through a typed data structure.
- **Module 06 (Not implemented):** Not implemented.
- **Module 07 (Gas Sponsor):** Supports gas sponsorship operations from an owners ZapWallet.
- **Module 08 (Not implemented):** Not implemented.
- Additional modules provide various other functionalities

Each module is identified by a unique AssetId and Address, ensuring secure and isolated operation.

2 Zap Stateless Predicate Wallet - "ZapWallet"

The Zap wallet enables stateless account abstraction through the use of secp256k1 elliptic curve cryptography. Let (sk, pk) be a key pair where sk is a private key and pk is its corresponding public key on the secp256k1 curve. For a ZapWallet predicate P, we define:

$$P : \mathbb{T} \times \mathbb{S} \to \{0, 1\}$$

where $\mathbb{T}$ is the set of valid transactions, $\mathbb{S}$ is the set of valid signatures, and the output $\{0, 1\}$ represents validation success or failure.

The "owner" of the predicate wallet possesses the private key sk used to generate signatures $\sigma \in \mathbb{S}$ that validate transactions spending UTXOs at the ZapWallet master predicate address. Thus, account abstraction is achieved by mapping:

$$f : (sk, pk) \to P$$

where f is the function that associates the key pair with the predicate's validation logic, enabling stateless control of the wallet's assets through standard elliptic curve signatures.

To enable diverse transaction types and functionality, the predicate P coordinates with a set of module predicates $\mathbb{M}$, where:

$$\mathbb{M} = \{M_0, M_1, ..., M_8\}$$

Each module M_i is identified by a unique AssetId and predicate address pair:

$$M_i = (a_i, p_i) \text{ where } a_i \in \mathbb{B}_{256}, p_i \in \mathbb{A}$$

where $\mathbb{B}_{256}$ is the set of 256-bit values and $\mathbb{A}$ is the set of valid addresses. The predicate P ensures that exactly one module is active in any valid transaction, except during initialization and upgrades, maintaining the wallet's state integrity.

2.1 AssetId Calculation

A ZapWallet requires unique AssetIds for each module and a single associated nonce asset. These AssetIds must be precalculated and provided to the master predicate during rehydration via the configurable code block.

2.1.1 Formula Definition

For any module or nonce asset, the AssetId is calculated using a two-step hashing process:

Let a be a padded EVM address, k be a key, and c be the ZapManager contract ID. Then:

$$h_1 = \text{SHA256}(a \parallel k)$$
$$h_2 = \text{SHA256}(c \parallel h_1)$$

Where:

- $\parallel$ denotes concatenation
- SHA256 is the SHA-256 hash function
- the final calculated AssetID $= h_2$

2.2 Constants

2.2.1 Key Constants

Each module and nonce has an associated key constant used in AssetId calculation.
Module Keys. For each module i, its key k_i is defined as:

$$k_0 = \texttt{0x0000000000000000000000000000000000000000000000000000000000000000}$$
$$k_1 = \texttt{0x0000000000000000000000000000000000000000000000000000000000000001}$$
$$\ldots$$
$$k_n = n \in \mathbb{F}_{2^{256}}$$

Nonce Key. For nonce assets, the key k_n is defined as:

$$k_n = \texttt{0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff}$$

These keys are used as distinct inputs in the AssetId calculation process, ensuring unique AssetIds for each module and nonce token.

2.3 Address Padding

EVM addresses must be padded to 32 bytes. For an address a_{evm}:

$$a = 0^{12} \parallel a_{evm}$$

where 0^{12} represents 12 zero bytes.

2.4 Example Calculation

Input Values

```
EVM Address (20 bytes):
ff03ffd5d3e881c60a91eaa30c67d03aec025c49

Padded EVM Address (32 bytes):
0000000000000000000000ff03ffd5d3e881c60a91eaa30c67d03aec025c49

ZapManager Contract ID:
c4442b787992c3afa14c0bfdec61b2921192e87494b226829c2d276ab855fc19
```

Calculation Steps—— Step 1: Calculate h_1 Concatenate the padded address with the key and apply SHA256:

```
h1 = SHA256(
    0000000000000000000000ff03ffd5d3e881c60a91eaa30c67d03aec025c49 ||
    ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
)
```

Step 2: Calculate h_2 Concatenate the ContractId with h_1 and apply SHA256:

```
h2 = SHA256(
    c4442b787992c3afa14c0bfdec61b2921192e87494b226829c2d276ab855fc19 ||
    h1
)
```

2.5 Master Predicate Configurables

The calculated Module AssetIds, Module Addresses and Owner Address must be provided to the master predicate through the configurables:

```
configurable {
    // Module 0 (Upgrade Module)
    ASSET_KEY00: b256 = <calculated_module_asset_id_0>,
    MODULE00_ADDR: Address = <module_0_predicate_address>,

    // Module 1
    ASSET_KEY01: b256 = <calculated_module_asset_id_1>,
    MODULE01_ADDR: Address = <module_1_predicate_address>,
    // ... modules 2-8

    // Owner Address
    OWNER_ADDRESS: b256 = <padded_evm_owner_address>,
}
```

2.6 Security Requirements

Uniqueness For any two modules i, j where $i \neq j$:

$$\text{AssetID}_i \neq \text{AssetID}_j$$

Determinism For any given inputs (a, k, c), the calculation must produce the same AssetID across all implementations of the same version of the ZapWallet:

$$f(a, k, c) = \text{AssetID}$$

where f is the AssetId calculation function.

3 Zap Manager Contract - "ZapManager"

3.1 Overview

The ZapManager contract facilitates the initialization and upgrade lifecycle of a ZapWallet through asset management and state control. It serves as the central authority for:

- Wallet initialization through controlled asset minting
- Upgrade path management between wallet versions
- State tracking through unique nonce assets
- Module asset distribution and verification

3.2 State Variables

The contract maintains the following state:

- $owner \in Address$: Contract administrator with privileged access
- $v1\_map : key1 \to AssetId$: Nonce asset tracking where:

$$key1 = sha256(evm\_addr || master\_addr)$$

- $can\_initialize \in \{true, false\}$: Initialization phase flag
- $can\_upgrade \in \{true, false\}$: Upgrade phase flag
- $v1\_version, v2\_version \in String[5]$: Version identifiers

3.3 Contract Phases

Contract state evolves through distinct phases defined by the tuple $(can\_initialize, can\_upgrade)$ where:

$$(can\_initialize, can\_upgrade) \in \{(false, false), (true, false), (false, true)\}$$

These phases represent:

- $(false, false)$: Initial or paused state
- $(true, false)$: Active initialization phase
- $(false, true)$: Active upgrade phase

Phase transitions are controlled by the contract owner and must maintain the invariant:

$$\neg(can\_initialize \land can\_upgrade)$$

3.4 Core Functions

3.4.1 Wallet Initialization

```
initialize_wallet(master_addr, owner_evm_addr, initdata) -> EvmAddress
```

Preconditions:

- $\neg is\_paused()$
- $can\_initialize = true$ (for InitModules)
- $key1 \notin domain(v1\_map)$ (for InitModules)

Effects:

- InitModules: Mints $(n+1)$ assets where $n = |modules|$ where:

$$\forall i \in [0, n].balance(asset_i, recipient_i) = 1$$

- NewModule: Mints single module asset where:

$$key \neq KEY\_NONCE \land balance(asset, recipient) = 1$$

Post-conditions:

- For InitModules:

$$key1 \in domain(v1\_map) \land balance(v1\_map[key1]) = 1$$

- For NewModule:

$$balance(new\_module\_asset) = 1$$

3.4.2 Wallet Upgrade

```
upgrade(owner_evm_addr, sponsored)
```

Preconditions:

- $\neg is\_paused()$
- $can\_upgrade = true$
- $\exists$ nonce asset in inputs where:

$$key1 = sha256(owner\_evm\_addr||nonce\_owner)$$

$$v1\_map[key1] = nonce\_asset\_id$$

Effects:

- Verifies nonce asset ownership:

$$balance(nonce\_asset\_id, nonce\_owner) > 0$$

- Records upgrade status
- Processes payment based on $sponsored$ flag

3.5 Asset Management

Asset balances and ownership are tracked through the balance function:

$$balance : AssetId \times Address \to \mathbb{N}$$

Key generation for nonce asset tracking:

$$key1 = sha256(evm\_addr||master)$$

Asset invariants must maintain:

- Nonce uniqueness:

$$\forall k_1, k_2 \in domain(v1\_map).k_1 \neq k_2 \implies v1\_map[k_1] \neq v1\_map[k_2]$$

- Balance consistency:

$$\forall k \in domain(v1\_map).balance(v1\_map[k]) > 0$$

3.6 Security Invariants

The security properties of the ZapManager contract are defined by the following formal invariants, which must hold true at all times during contract execution:

3.6.1 Asset Mapping Integrity

For any key in the nonce asset mapping:

$$\forall k.v1\_map[k] \neq \emptyset \implies balance(v1\_map[k]) > 0$$

This invariant ensures that any nonce asset recorded in the mapping must maintain a positive balance. This is critical for:

- Preventing double initialization of wallets
- Maintaining unique wallet identities
- Ensuring valid upgrade paths

3.6.2 Phase Exclusivity

The contract phases must remain mutually exclusive:

$$\neg(can\_initialize \land can\_upgrade)$$

This enforces strict separation between initialization and upgrade phases, which:

- Prevents state confusion
- Ensures clean wallet lifecycle progression
- Maintains clear operational boundaries

3.6.3 Administrative Control

The contract must maintain valid ownership:

$$owner \neq \emptyset$$

This invariant guarantees:

- Continuous administrative control
- Emergency intervention capability
- Proper governance of contract parameters

3.6.4 Asset State Consistency

For any wallet w with nonce asset n and module assets M:

$$balance(n) \in \{0, 1\}$$

$$\forall m \in M.balance(m) \leq 1$$

These balance constraints ensure:

- Unique wallet identification through nonce assets
- Proper module asset distribution
- Prevention of asset duplication

3.6.5 Verification

These invariants are maintained through:

1. Runtime checks in all state-modifying functions
2. Access control restrictions on administrative operations
3. Balance verification during initialization and upgrades
4. Event emission for off-chain monitoring

3.6.6 Events

Event emission follows state transitions:

- InitializeWalletEvent(master_addr, owner_evm_addr, is_base_modules)
- UpgradeEvent(owner_evm_addr, master_address, is_sponsored, verified_nonce)
- ContractStateEvent(allow_initialize, allow_upgrade, sender)
- WalletVersionsEvent(v1_version, v2_version, sender)

4 Zap Master - `"master"`

4.1 Overview

The ZapWallet Master is a predicate that provides secure asset custody through modular validation. The master serves as the authority to validate ZapWallet transactions, initialization (if paying own gas) and wallet upgrades.

4.2 Definitions

Basic Types
Let:

- $\mathbb{B}_{256}$ be the set of 256-bit values
- $\mathbb{A}$ be the set of valid Fuel addresses
- $\mathbb{U}_{64}$ be the set of 64-bit unsigned integers

Module
A module M is defined as a tuple:

$$M = (a, p) \text{ where } a \in \mathbb{B}_{256}, p \in \mathbb{A}$$

where:

- a is the unique AssetId of the module
- p is the predicate address of the module

Wallet Configuration
A wallet configuration W consists of:

$$W = (M_0, M_1, ..., M_8, o)$$

where:

- M_0 is the upgrade module
- M_1 through M_8 are the operational modules
- $o \in \mathbb{B}_{256}$ is the owner's address

4.3 Transaction Types and Validation

The ZapWallet master predicate's `main()` function accepts an optional `WalletOp` parameter that serves two distinct purposes depending on the transaction type:

1. **Initialization:** When present (`Some(WalletOp)`), validates wallet initialization
2. **Module Operations:** When absent (`None`), validates module-based transactions

4.3.1 Initialization

During initialization, `WalletOp` contains:

```
pub struct WalletOp {
    pub evm_addr: b256,      // padded ETH address of wallet owner
    pub compsig: Bytes,      // Compact signature
    pub command: String,     // Initialization command
}
```

An initialization transaction T_i must satisfy:

$$V_i(T_i, s, o) = true$$

where:

- s is a valid EIP712 signature
- o is the owner's address
- V_i is the initialization verification function

V_i verifies:

1. Exactly two inputs: one coin and one contract
2. Valid change output
3. Valid signature recovering to owner
4. No module assets present

4.3.2 Module Operations

For all other operations, transaction validation relies on:

- Presence of exactly one module asset in inputs
- Proper return of module asset in outputs
- Module-specific validation logic
- Nonce token accounting (when required)

A module operation transaction T_m must satisfy:

$$V_m(T_m, M_i) = true$$

for exactly one module M_i where $i \in \{1, ..., 8\}$
V_m verifies:

1. Exactly one module asset present in inputs
2. Module asset returned correctly in outputs
3. Module-specific validation passes

Upgrade Operation
An upgrade transaction T_u must satisfy:

$$V_u(T_u, M_0) = true$$

V_u verifies:

1. Only module M_0 present
2. Upgrade module asset handled correctly (sent to ZapManager)
3. No other module assets present

This dual-purpose design allows the `WalletOp` to handle initialization while remaining unintrusive for regular module operations.

4.4 State Transitions

Module State Vector
For any transaction T, let $\vec{m}$ be a boolean vector where:

$$\vec{m} = [m_0, m_1, ..., m_8] \text{ where } m_i \in \{0, 1\}$$

indicating the presence (1) or absence (0) of each module in the transaction inputs.

Valid States
A transaction is valid if and only if $\vec{m}$ satisfies exactly one of:

1. Initialization: $\vec{m} = [0, 0, ..., 0]$
2. Single Module: $\vec{m}$ contains exactly one 1
3. Upgrade: $\vec{m} = [1, 0, ..., 0]$

4.5 Security Properties

Module Isolation
For any valid transaction T:

$$|\{i : m_i = 1\}| \leq 1$$

Meaning no more than one module can be active in a single transaction.

Asset Conservation
For any module M_i present in transaction inputs:

$$\exists \text{ output } o : o.asset = M_i.a \land o.amount = 1 \land o.to = M_i.p$$

Meaning any module asset must be properly returned to its predicate.

State Consistency
The master predicate enforces:

- No double-module usage
- Proper initialization sequence
- Upgrade isolation
- Module asset conservation

4.5.1 Implementation Notes

Module Detection
The system identifies modules by matching both:

- AssetId ($\mathbb{B}_{256}$)
- Predicate Address ($\mathbb{A}$)

Validation Flow
1. Scan inputs for module assets 2. Build module state vector 3. Determine transaction type 4. Apply appropriate validation rules 5. Verify output conditions

4.6 Initialization flow

The following outlines the procedure to construct the InitData struct and build the initialization transaction that calls the ZapManager contract to mint nonce tokens.

1. To obtain the nonce token id a user can calculate it on "by-hand" by the above method, or, obtain by calling the above zap JSON-RPC method `zap_get_assetidKey1()` with the EVM address of the owner.

2. The nonce token AssetId (32byte value) is Signed by the use via connected wallet (if using the Zap frontend App) or by an appropriate Ethereum library (like ethers-rs) and sent back as hex encoded bytes to the rpc as a compact signature.

3. The zap JSON-RPC method `zap_get_initializationTxid()` is called with the signers EVM address and compact signature from (2) as a parameters.

4. The return value from (3) is the initialization transaction id. The user is then required to sign the 32-byte transaction id and return it the compact signature to the zap rpc. The two options for this are 1; by the Zap frotnend using an EVM browser wallet extension that is connected, or, 2; with an appropriate Ethereum library (like ethers-rs).

5. The zap JSON-RPC method `zap_SubmitInitializationTx()` is called with two parameters, the compact signature from (4) and the EVM signers address. This submits the initialization transaction to the Fuel node.

6. Once (5) is complete. transaction success can be checked by calling the eth JSON-RPC method `eth_getTransactionReceipt()`.

4.7 Initialization Flow

The initialization of a ZapWallet can be done using the owner Fuel BASE_ASSET to pay for gas or from a third party that spends their own gas. If the owner is initializing their own ZapWallet (self-initialization) which spends a BASE_ASSET UTXO from the master, this involves a signature within the `WalletOp` parameter.

4.7.1 Transaction Structure

A valid initialization transaction requires:

- Exactly two inputs:
  - One coin input containing FUEL BASE_ASSET for gas
  - One contract input referencing the ZapManager contract
- A single change output returning unused FUEL BASE_ASSET to the transaction sender

4.7.2 Signature Requirements

The self-initialization route initialization process requires a single distinct signature:

1. **Transaction Signature:** A standard EIP-712 compliant signature over the initialization data structure:

```
Initialization(
    string command,    // "ZapWalletInitialize"
    bytes32 evmaddr,   // Owner's ETH address
    bytes32 utxoid     // UTXO ID of the coin input
)
```

4.7.3 Contract Interaction

The initialization flow proceeds as follows:

1. The transaction calls the ZapManager contract initialize_wallet() function
2. The contract verifies the nonce has not already been minted
3. Upon verification, the contract:
   - Mints the nonce tokens
   - Sends (NONCE_MAX - 1) to the ZapWallet master predicate
   - Retains one token for transaction validation
4. The master predicate verifies:
   - The initialization signature
   - Transaction structure
   - Change output validity

Upon successful completion of the initialize_wallet() function call, the ZapWallet is initialized and ready for module operations.

5 Module 00 - "module00_upgrade"

spec writeup WIP.

6 Module 01 - "module01_evm_txtype1"

spec writeup WIP.

7 Module 02 - `""module02_evm_txtype2"`

spec writeup WIP.

8 Module 03 - `""module03_erc20"`

spec writeup WIP.

9 Module 04 - `""module04_txidwit"`

spec writeup WIP.

10 Module 05 - `""module05_eip712_simple"`

spec writeup WIP.

11 Module 06 - `""module06_eip712_contract"`

spec writeup WIP.

12 Module 07 - `""module07_sponsor"`

spec writeup WIP.

13 Module 08 - `""module08"`

Not implemented in v0.8.0.