Specification

Infrastructure for Stateless Account Abstraction on Fuel

Date: April 2024

1 Introduction

The following specification is broken down into two parts. The Predicate wallet written in Sway and the backend JSON-RPC api commands.

Predicate Wallet:
The predicate wallet is a stateless account abstraction wallet written in Sway. The ”wallet” contains the logic to validate a specific transaction types signed by the owner of the wallet.

Backend:
The backend consists of an RPC endpoint that accepts `eth_`,`net_` and `zap_` namespace calls and a project (that is required to build the RPC) called the ”zap-executor”, which builds and sends transactions to Fuels network, implements the interface from an API call to the graph-ql port of fuel-core and any translation to evm spec data structures.

2 Predicate Wallet

Zaps predicate wallet enables stateless account abstraction, by way of secp256k1 private keys ability to sign transactions that spend UTXOs at the predicate wallet. The ”owner” of the predicate wallet is also the owner of the public/private key pair used to sign the any transaction(s) to the wallet itself. Thus account abstraction is obtained from the original public/private key.

2.1 Validation types

There are currently two transaction types that can be supplied to the predicate wallet within the data structure `PValidation`.

`tx_type`:

0. Initialization transaction.
1. EVM signed transaction (Legacy, EIP-1559).

The data struct `PValidation` is the only data that is passed into the main() function of the predicate wallet.

```
pub enum SuppliedData {
    InitData: Init,          // type (0)
    SignedData: Bytes,       // type (1)
}

struct PValidation {
    tx_type: u64,
    data: SuppliedData,
}
```

If the `tx_type` is equal to zero (0), `PValidation` contains `InitData` within the `SuppliedData` struct.

```
pub struct Init {
    signature_owner: B512,      // signature of the Assetid.
    key: b256,                  // the Key, for nt AssetId.
    wit_index: u64,             // witness index.
}
```

`Init` is a data struct containing the compact signature (`signature_owner`) of the nonce token asset id that will be minted by the nonce manager contract within the current transaction. The key (`key`) that was used to create the nonce token asset id (see section **??**) and the witness index (`wit_index`) of the witness to the current transaction.

If the `tx_type` is equal to one (1), `PValidation` contains `SignedData` which is of type Bytes. The Bytes in this case is an RLP encoded EVM typed transaction.

2.1.1 Validation logic type 0

The purpose of `tx_type` = 0 is to provide the validation logic for a transaction that initializes the predicate wallet. This consists of calling the nonce manager contract which mints ”nonce token” assets and sends the mint amount - 1 to the predicate wallet. Once the predicate wallet contains some amount of this asset type, and the nonce manager contract contains at least 1 (one) of these assets, the predicate wallet is considered initialized.

The `tx_type` = 0 transaction type is necessary to mint ”nonce token” assets, without nonce tokens of a specific asset_id held as a UTXO at the predicate wallet address, `tx_type` = 1 transactions will not work. Within an EVM Legacy and EIP-1559 transaction the ”nonce” field is populated with the current transaction count for the account (EOA) making the transaction call. This number (the nonce) is incremented every time a transaction is successfully included in a block on an EVM

compatible chain. However when constructing a predicate wallet (abstracted account) there is no concept built in to a transaction type that includes the facility to obtain a transaction count for a particular address or predicate root. Therefore, for a predicate wallet to validate EVM style transactions, that are presented to the predicate as a simple byte stream, there must be a some mechanism by which a transaction count (for the predicate wallet) can persist past the scope of a single transaction. Nonce tokens serve the purpose of keeping track of the transaction count of a predicate wallet. For each successful transaction (EVM style) the nonce token count is reduced by one. That is, for every transaction, the total amount of nonce tokens (held as a UTXO) are included as an input to the transaction and the total amount - 1 is included as a transaction output to the same transaction. Thus giving the transaction builder an easy mechanism to obtain the current transaction count (nonce) of a predicate wallet.

To calculate the nonce (token) AssetId of the following example is given taking the variables:

```
32 byte "the key" or sub_id,
0000000000000000000000000000000000000000000000000000000000000001

32 bytes padded evm address,
000000000000000000000000ff03ffd5d3e881c60a91eaa30c67d03aec025c49

32 bytes the NonceManager3's own ContractId, (beta-5)
c4442b787992c3afa14c0bfdec61b2921192e87494b226829c2d276ab855fc19


The AssetId is calculated by this by:
h1 = sha256("padded_evm_address":"key")
h2 = sha256("contractid":"h1")
AssetId = h2
```

example:

```
sha256(
000000000000000000000000ff03ffd5d3e881c60a91eaa30c67d03aec025c49,
0000000000000000000000000000000000000000000000000000000000000001)
=

8a1aa9c4f2b55e2991a44f390d27c12617b2a9b0e9d48546f06b709374f3dfd8
```

```
sha256(
c4442b787992c3afa14c0bfdec61b2921192e87494b226829c2d276ab855fc19,
8a1aa9c4f2b55e2991a44f390d27c12617b2a9b0e9d48546f06b709374f3dfd8)
=

280ffeb1249eeb3807bb24c2f5d9d71f438e81e6b67145c9a00a3ae76f7d3a5d
```

Nonce token AssetId that will be minted:

```
280ffeb1249eeb3807bb24c2f5d9d71f438e81e6b67145c9a00a3ae76f7d3a5d
```

The following outlines the procedure to construct the InitData struct and build the initialization transaction that calls the Zap Nonce Manager 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()`.

2.1.2 Validation logic type 1

The purpose of `tx_type` = 1 is to provide the validation logic for a signed typed RLP encoded EVM transaction for the base asset (ETH) on Fuel.

The current predicate wallet supports Legacy and EIP-1559 EVM transactions. Both EVM transaction types are decoded by the predicate wallet itself and transaction criteria are matched for validity.

EVM Legacy Tx:

The following example best explains how the `tx bytes rlp` from a Legacy EVM transaction are encoded and decoded within `rlp_utls5.sw`.

transaction bytes encoded:

```
eb08822cb482520894ff03ffd5d3e881c60a91eaa30c67d03aec025c4987ea7aa
67b2d00008083097bc88080 + compact signature


eb --> 0xc0 + length of payload (everything before the sig,
        not including the first byte ) 0xc0 + 0x2b = 0xeb
08 --> nonce
82 --> rlp encoded integer (0x80 and 0x02 bytes long)
2cb4 --> gas_price
82 --> rlp encoded integer (0x80 and 0x02 bytes long)
5208 --> gas_limit or max_fee_per_gas
94 --> rlp encoding
ff03ffd5d3e881c60a91eaa30c67d03aec025c49 --> to (evm address)
87 --> (0x80 and 0x07 bytes long)
ea7aa67b2d0000 --> value (amount wei)
8030
97bc8 --> chain_id --> odd number of characters
8080
```

The logic for the above decoding can be seen in `rlp_utls5.sw` from lines 153-280

EVM EIP-1559 Tx:

In a similar way the following example best explains how the `tx bytes rlp` from an EIP-1559 EVM transaction are encoded and decoded within `rlp_utls5.sw`.

transaction bytes encoded:

```
02f87283097bc8018085011bd4e67982520894ff03ffd5d3e881c60a91eaa30c6
7d03aec025c49881148e7a6abbfc00080c001a0 + compact signature
```

```
02 --> tx type
f872 --> rlp encoding byte length
83 --> rlp encoded 3 bytes
097bc --> chain id
80 --> rlp encoded 1 byte
1 --> nonce
8085 --> rlp encoded 5 bytes
011bd4e679 --> gas price
82 --> rlp encoded 2 bytes
5208 --> gas limit
94 --> rlp encoded 20 bytes
ff03ffd5d3e881c60a91eaa30c67d03aec025c49 --> to (evm address)
88 --> rlp encoding 8 bytes
1148e7a6abbfc000 --> value (wei)
80 --> rlp encoded 1 byte
c0
01a0 --> rlp encoded
```

The logic for the above decoding can be seen in `rlp_utls5.sw` from lines 72-152

Tx validation:

At the current commit a Legacy or EIP-1559 EVM transaction is validated on the following conditions:

1. The transaction is signed by the owner of the predicate wallet (secp256k1 private key).
2. The transaction contains a nonce is equal to (0xFFFFFFFF - nonce) remaining nonce tokens held in the predicate wallet.
3. There is at least one one output for the remaining nonce tokens equal to (0xFFFFFFFF - (nonce -1)) and returned to the predicate wallet.

Note: currently no checks are done against the value sent within the transaction or the change value. However, when the transaction is submitted to the RPC there are change outputs added, and a failure will occur if the EVM signed transaction amount is greater than the predicate wallet balance.

3 JSON-RPC methods

Zaps JSON-RPC supports the following methods for `eth_`,`net_` and `zap_` namespace calls.

3.1 `eth`

`_chainId`
Returns the current network/chain ID, used to sign replay-protected transaction introduced in EIP-155.

`_blockNumber`
Returns the latest block number of the blockchain.

`_getBlockByNumber`
Returns information of the block matching the given block number.

`_getBlockByHash`
Returns information of the block matching the given block hash.

`_getBalance`
Returns the balance of given account address in wei.

`_feeHistory`
Returns the collection of historical gas information.

`_gasPrice`
Returns the current gas price on the network in wei.

`_getCode`
Returns the compiled bytecode of a smart contract.

`_estimateGas`
Returns an estimation of gas for a given transaction.

`_call`
Executes a new message call immediately without creating a transaction on the block chain.

`_getTransactionCount`
Returns the number of transactions sent from an address.

`_sendRawTransaction`
Creates new message call transaction for signed transactions.

`_getTransactionReceipt`
Returns the receipt of a transaction by transaction hash.

`_maxPriorityFeePerGas`
Get the priority fee needed to be included in a block.

`_getTransactionByHash`
Returns the information about a transaction from a transaction hash. (note currently will not work for Legacy tx)

3.2 zap

`_getPredicateAddress`
returns the address of the predicate wallet on Fuel, for the EVM address.

`_checkInitialized`
returns true or false if the predicate wallet for the EVM address has been initialized.

`_get_assetidKey1`
Returns the nonce token asset id, for the predicate wallet for key = 1.

`_get_initializationTxid`
Returns the transaction id of the initialization transaction for the predicate wallet.

`_submitInitTx`
Submits the initialization transaction to the Fuel node.

`_getTransactionCount`
Returns the number of transactions made by the predicate wallet.

3.3 net

`_version`
Returns the current network id.