Laser Wallet Specification

V 1.0.0

March 5, 2022

***First Draft - Not Complete**

PART 1. General Overview

I. Introduction

Laser Wallet is a flexible multi - signature Ethereum smart contract based wallet. It implements account abstraction (EIP4337) allowing the wallet to behave like a regular account. The wallet's EOA keys are generated and stored on the mobile device.

II. Security

The core of Laser Wallet's code is a modification from Gnosis Safe, a battle-tested multi-signature wallet. We also have audits from: **** INSERT AFTER AUDIT ****

III. Flexibility

Currently, there are primarily three types of wallets:

1. **Software EOA - >** Software Externally Owned Accounts basically just handle your key pair inside of your mobile or browser extension.

2. **Hardware Wallets - >** Hardware Wallets handle your key pair in a segregated manner.

3. **Smart Contract Wallets - >** Smart Contract Wallets are each programmed uniquely, as they have more programmability capabilities.

The issue with software based EOA’s is that all the responsibility lies in the user holding the private key, and they are more likely to get hacked due to the fact that they are more exposed.

Hardware Wallets are much safer because the keys are completely isolated inside of the hardware, the problem is that they are not user friendly (you need one more device), and all the responsibility lies on the user holding the device (and seed).

Our solutions for solving these pain points are:

1. **Mobile-first mindset:** Around 84% of the world’s population own a smart phone. We want to leverage this momentum by allowing users to be part of Defi and crypto in general. By focusing mobile, users are one click away to create a wallet and join the new world.

2. **Flexible multi - sig:** We want to leverage the security that multi - signature wallets offer but also add flexibility to it. That is why we introduced the ***Special Owners***.

3. **Account Abstraction:** Laser Wallet is EIP4337 compliant, this means that the smart contract wallet “behaves” on a high level, like a regular EOA (more on this later).

IV. Special Owners / Owners

A common multi - signature wallet consists of a set of owners and a threshold. The threshold is the minimum number of signatures “authorizations” required to execute a transaction.

For example:

If there are 3 owners: Alice, Bob, and Charlie. And a threshold of 2, in order to execute any transaction, 2 owners need to authorize it.

If Alice wants to do a transaction, it needs to wait for the signature of either Bob or Charlie. Respectively, this also applies to Bob and Charlie.

The benefit of this design is that it eliminates a single point of failure (if Alice gets hacked, nothing happens because her

signature by itself is useless). The downside, is that it is very slow and inconvenient.
Following the same example, if Alice is a user that does multiple transactions a day, she needs to wait every time for a second authorization making the whole process slow and frustrating.

These are the reasons why we introduced Special Owners.

Special Owners are a type of owners that can execute any transaction with only their signature. Going back to our example where there are 3 owners: Alice, Bob and Charlie and a threshold of 2. If Alice is a Special Owner, she can do any transaction without requiring further approvals, skipping the threshold requirement.

Regular Owners "Owners" can authorize transactions, but each signature is worth "1". In order to execute any transaction, the required threshold must be met.

In diagram 1.0 you will find a simple example of a wallet with a threshold of 2. In the example, you will see that in order to execute a transaction, either: a) the special owner signs it or b) two regular owners sign it (two owners because the threshold is 2).

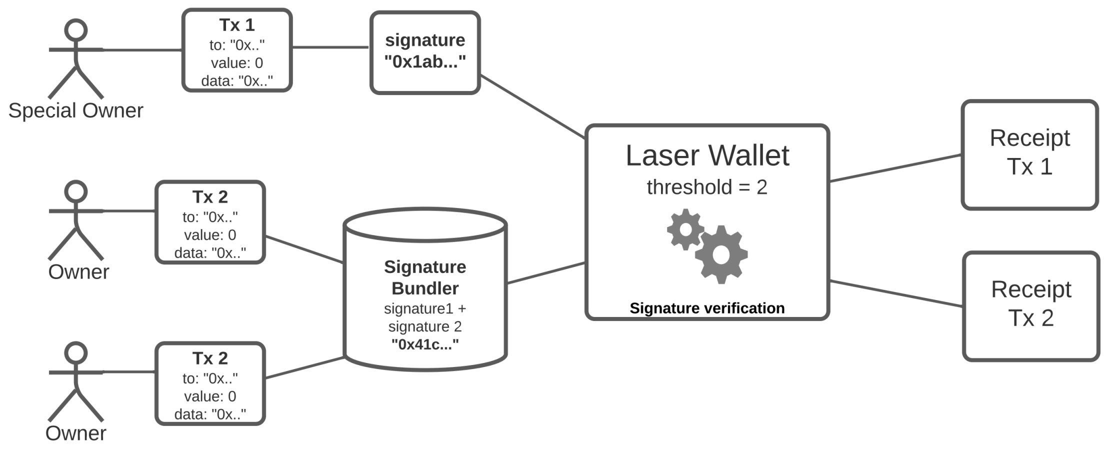


Diagram 1.0

This design system has many benefits, and can be used for many possible scenarios. Below are just a few:

1. **Single user + backup:** Solo users usually just have a regular EOA account. As we previously discussed, if the user looses the keys, the account is gone forever. With Laser Wallet, a user can make himself a Special Owner to have the benefits of an Externally Owned Account (making fast transactions) but also the added security of having more owners in case the private key is lost. Of course, if someone takes hold of the private key of a Special Owner, the whole wallet is in danger. But in the case that the private key is just lost, there is a way to recover the account, making the other owners act like "guardians".

2. **Shared Wallet:** Users can have a shared wallet (like a family account). Where there are multiple special owners, but all the family keeps track of the transactions.

3. **Startup wallet:** Startups can have a specific configuration, as for example, having multiple owners with a threshold, and giving the owner or CTO a special owner account.

4. **Regular Multi-Sig:** Users can also choose to not having a special owner, and take the more standard approach of having a general multi-sig.

These are just a couple of examples, the users can choose to configure the wallet however they like.

PART 2. Smart Contracts Specification

I. Transactions

For the wallet, a transaction is any call that changes the state of the chain. There are two types of transactions:

1. **Internal transactions:** These transactions change something about the wallet's configuration. It can be changing the threshold, adding or removing an owner, etc…

2. **External transactions:** These transactions send value or calldata to an external source. It can be to send Eth, send an ERC-20 token, swap a token in Uniswap, stake Eth in RocketPool, etc…

Every transaction, either internal and / or external, can only be executed through the ***execTransaction(…)*** function in ‘**LaserWallet.sol**’. *With one exception (EIP4337) that we will cover later on.

```
function execTransaction(
        address to,
        uint256 value,
        bytes calldata data,
        Enum.Operation operation,
        uint256 safeTxGas,
        uint256 baseGas,
        uint256 gasPrice,
        address gasToken,
        address payable refundReceiver,
        bytes memory signatures,
        address specialOwner
    ) public payable virtual returns (bool success)
```

LaserWallet.sol

For internal transactions, the address of the wallet needs to be provided in the first parameter “to”. This is because all of the wallet’s internal configurations (like changing an owner or the entry point address) are protected by an “authorized” modifier:

```
function addOwnerWithThreshold(address owner, uint256 _threshold)
        public
        authorized
```

OwnerManager.sol

The authorized modifier restricts the access to everyone but the wallet itself(address(this)).

```
function requireSelfCall() private view {
        require(msg.sender == address(this), "Only callable from the wallet");
    }

    modifier authorized() {
        requireSelfCall();
        _;
    }
```

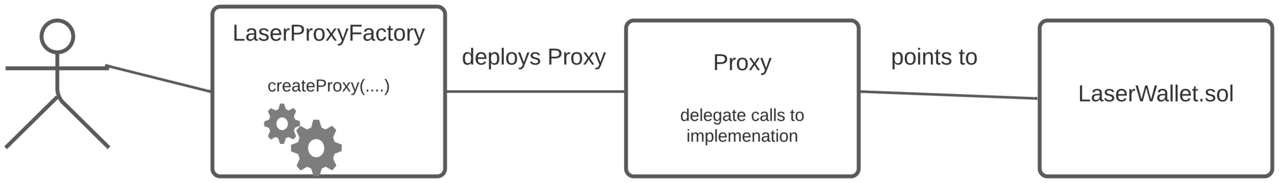
SelfAuthorized.sol

For external transactions, the address of the recipient is provided (like a regular transaction).

NOTE: An internal transaction can also be executed by calling another contract via a “delegatecall”.

II. Smart Contracts General Architecture

Laser Wallet uses the proxy standard to minimize deployment costs.



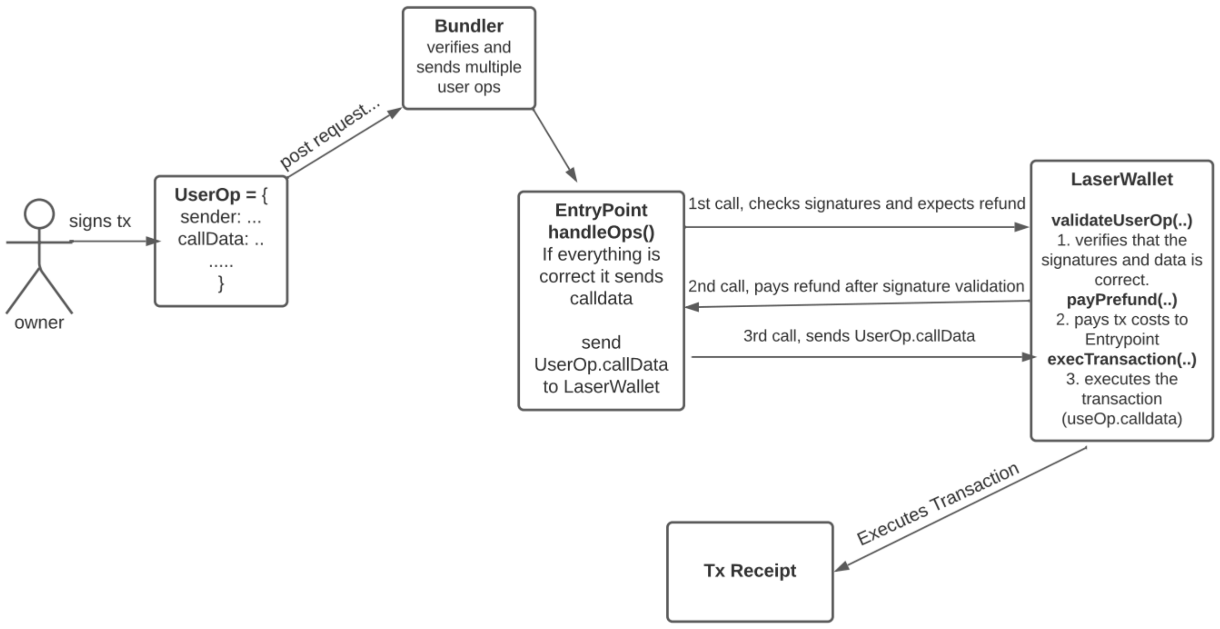
Every time a user is creating a new wallet, it creates a new proxy contract that delegate all calls to a base contract (“LaserWallet.sol”). For the purposes of our contracts, the implementation contract address is called “singleton” and it is located at storage slot 0.

* For security reasons, the wallet needs to be initialized in the same transaction that it is created. In order to accomplish this, a call to function **setup(..)** in "LaserWallet.sol" needs to be sent as calldata in the proxy factory creation.

Possible ways to send transaction info

There are three possible ways to send the signatures of the owners with its data to the wallet (to make a transaction).

1. **EIP4337:** EIP4337: **https://eips.ethereum.org/EIPS/eip-4337** allows a wallet to "behave" like a regular EOA. There are many benefits this can bring to smart contract wallets. But for our purposes, it allows the smart contract wallet to pay gas fees for itself. This is done by sending the transaction information (UserOperation) to a "bundler" through a post request. The bundler then does some verifications, and then sends the UserOperation to a contract called "EntryPoint". The EntryPoint contract then calls a function in the wallet called "**validateUserOp**". The main function of validateUserOp is 1) to check that the information is correct (check the signatures and tx data) and 2) pay the gas fees to the EntryPoint contract. Once this is done, EntryPoint sends the calldata to the wallet, and the wallet then executes whatever the transaction is. The big benefit of this design is that users won't have to have Eth in their EOA to pay for the gas fees. And in future versions, we will be able to implement quantum-proof signatures schemes.



Bundler
verifies and sends multiple user ops
post request...
signs tx
owner
UserOp = { sender: ... callData: .. .. }
EntryPoint handleOps()
If everything is correct it sends calldata
send UserOp.callData to LaserWallet
1st call, checks signatures and expects refund
2nd call, pays refund after signature validation
3rd call, sends UserOp.callData
LaserWallet
validateUserOp(..)
1. verifies that the signatures and data is correct.
payPrefund(..)
2. pays tx costs to Entrypoint
execTransaction(..)
3. executes the transaction (useOp.calldata)
Executes Transaction
Tx Receipt

2. **Off-chain signatures:** With off-chain signatures, the signatures of the owners are stored off-chain, and when the final owner approves the transaction, it sends the information on-chain. The big drawback with this approach, is that the last owner needs to have gas fees in his/her EOA, in order to pay the transaction fees.

3. **100% On-chain:** There is also a way to do everything on-chain. The solution is to get the hash of the transaction and call the **approveHash(..)** function (only an owner can call the function). This solution is not very efficient because there is a cost every time an owner wants to approve a transaction.

III. Upgradability

As reviewed in the "Smart Contracts General Architecture" section, the wallet is a proxy contract that delegates all calls to a base implementation contract.

Therefore, upgradability is possible by changing the implementation contract address. This is only possible through the **execTransaction(…)** function. The choice to upgrade the wallet relies 100% on the owners because the owners of the wallet are the only ones that can upgrade the wallet (change the implementation "singleton" address).

IV. Special Owners / Owners

By design, all special owners are also owners. **All the configuration to add / remove owners is in "OwnerManager.sol".**

When executing a transaction, there is an argument with the name of "specialOwner". If address(0) is given, then the wallet executes a normal transaction, requiring the minimum amount of approvals (threshold).

```solidity
function execTransaction(
        address to,
        uint256 value,
        bytes calldata data,
        Enum.Operation operation,
        uint256 safeTxGas,
        uint256 baseGas,
        uint256 gasPrice,
        address gasToken,
        address payable refundReceiver,
        bytes memory signatures,
        address specialOwner
    ) public payable virtual returns (bool success)
```

LaserWallet.sol

If an address != to address(0) is given, then the wallet assumes that the address is a special owner and sets the threshold to "1" only for the length of that transaction.

```solidity
// If a special owner is authorizing this transaction, then threshold is 1.
      uint256 _threshold = specialOwner != address(0) ? 1 : threshold;
```

It then does the correct verification to check if: a) the address provided is a special owner and b) the signature corresponds to the address.