Argent Documentation

Argent Team

February 28, 2019

1 Specifications

1.1 Introduction

The Argent wallet is an Ethereum Smart Contract based mobile wallet. The wallet's user keeps an Ethereum account (Externally Owned Account) secretly on his mobile device. This account is set as the owner of the Smart Contract. User's funds (ETH and ERC20 tokens) are stored on the Smart Contract. With that model, logic can be added to the wallet to improve both the user experience and the wallet security. For instance, the wallet is guarded, recoverable, lockable, protected by a daily limit and upgradable.

1.2 Guardians

The wallet security model is based on the ability to add Guardians. A Guardian is an account (EOA or smart contract) that has been given permission by the wallet's owner to execute certain specific operations on their wallet. In particular guardians can lock, unlock, and trigger a recovery procedure on the wallet as well as approve the execution of a large transfer to an unknown account.

We do not impose restrictions on who or what Guardians are. They can be can be a friend's Argent wallet, a friend's EOA, a hardware wallet, or even a paid third-party service.

Adding a Guardian is an action triggered by the wallet owner. While the first Guardian is added immediately, all subsequent additions must be confirmed after 24 hours and no longer then 36 hours after the addition was

requested. This confirmation windows ensures that a pending addition will be canceled (expire) should the wallet be locked or recovered.

Removing a Guardian is an action triggered by the wallet owner. It must always be confirmed after 24 hours and no longer then 36 hours after the removal was requested. This leaves the legitimate wallet owner enough time to notice and prevent the appointment of an illegitimate Guardian (or the dismissal of a legitimate Guardian) in case the owner lost control over their mobile device.

1.3 Locking

In case the wallet owner suspects his account (i.e. device) is compromised (lost, stolen, ...), he can ask any of his Guardians to lock the wallet for a security period of 5 days. Once the wallet is locked only a limited set of actions can be operated on the wallet, namely the recovery procedure, the unlock procedure, or the revocation of Guardians. All other operations (add guardian, assets transfer, ...) are blocked.

To unlock a wallet before the end of the security period, any guardian should trigger a wallet unlock.

1.4 Recovery

Wallet recovery is a process requested by a user who asserts ownership of a wallet. A successful recovery sets a new account as the wallet owner. This process should be validated by the wallet's guardians to be executed. Once a recovery has been executed it may be finalised after 36 hours, unless it has been cancelled.

The number of signatures (owner and/or guardians) needed to execute a recovery is given by

$$\left\lceil \frac{n+1}{2} \right\rceil$$

where n is the total number of guardians and $\lceil \rceil$ is the ceiling function.

A recovery can be cancelled before finalisation. The number of signatures (owner and/or guardians) needed to cancel a recovery is given by

$$\left\lceil \frac{n+1}{2} \right\rceil$$

where n is the total number of guardians when the recovery was executed.

Once a recovery is started the wallet is automatically locked. The wallet can only be unlock by finalising or cancelling the ongoing procedure, i.e. Guardians cannot unlock during a recovery.

1.5 Daily Transfer Limit

The wallet is protected by a daily limit (rolling for 24 hours). The owner can spend up to the daily limit in a given 24 hours period. The daily limit default value is 1 ETH and can be modified by the owner but it takes 24 hours for the new limit to be effective.

Any transfer exceeding the daily limit will be set as pending, and can be executed only after 24 hours.

Transfers to whitelisted addresses (see Section 1.6) and transfers approved by guardians (see Section 1.7) do not contribute to the daily limit.

The daily limit is cross-token (ETH + ERC20) and we're using an on-chain exchange (i.e. Kyber Network) to get the conversion rates for ERC20 tokens.

1.6 Whitelist

The wallet keeps a whitelist of trusted addresses. Transfers to those addresses are immediate and their amounts are not limited.

Adding an address to the whitelist is an action triggered by the wallet owner and takes 24 hours to be effective. Removing an address is triggered by the owner and is immediate.

1.7 Approved Transfer

Pending transfers exceeding the daily limit can be executed immediately by the owner, provided that he obtains signed approval from a number of guardians given by

$$\left\lceil \frac{n}{2} \right\rceil$$

1.8 ERC20 Exchange

The owner will be able to exchange ETH against ERC20 tokens. We're using an on-chain exchange (i.e. Kyber Network) to allow this. we (intend

to) charge a fee on each transaction.

1.9 ENS

The Wallet will be associated to an ENS. This association should be forward and backward meaning that it should be possible to obtain the Wallet address from the ENS and the ENS from the Wallet address.

1.10 Upgradability

The wallet must be upgradable to fix potential bugs and add new features. The choice of whether to upgrade or not a wallet will be left to the wallet owner. In particular, it should not be possible for a centralised party such as Argent to force a wallet upgrade and change an implementation that is assumed to be immutable by the owner.

1.11 ETH-less Account

In order to execute a wallet operation, owner and guardians should be able to use their account's private keys to sign a message showing intent of execution, and allow a third party relayer to execute them. Therefore their account don't pay transaction fees and don't need ethers (ETH-less account). In that case the wallet contract should be able to refund the gas (partially or totally) required to execute the transaction to the third party relayer. This pattern is similar to what is described in EIP 1077[1].

1.12 Authorized Third-Party Accounts

Wallet owners can authorize (and revoke at any time) specific third-party accounts to transfer ETH or call external contracts (optionally attaching ether to these calls) from their wallets. The wallet owner can specify a daily limit in ETH that applies across all third-party accounts. This functionality allows third-party Dapps to perform the aforementioned wallet operations without every time requiring a confirmation from the user. Authorizing (and revoking) a third-party account is always immediate.

[1] https://eips.ethereum.org/EIPS/eip-1077

	Lock/Unlock	Execute Recovery	Cancel Recovery	Approve Transfer
	Guardians	Owner OR Guardians	Owner OR Guardians	Owner AND Guardians
1	1	1	1	2
2	1	2	2	2
3	1	2	2	3
4	1	3	3	3
5	1	3	3	4

Table 1: Number of signatures required to perform operations. Depending the type of the operation, signatures can be required from guardians only or by a combination of guardians and/or owner.

2 Implementation

2.1 Smart Contracts architecture

Our architecture is made up of multiple contracts (see Figure 2.1). A first group of contracts form the infrastructure required to deploy or update user wallets. These infrastructure contracts are meant to be deployed only once (in theory):

- **Multisig Wallet:** Custom-made multi-signatures wallet which is the owner of most of other infrastructure contracts. All calls on those contracts will therefore need to be approved by multiple persons.
- **Wallet Factory:** Wallet factory contract used to create proxy wallets (in batches) and assign them to users.
- **ENS Manager:** The ENS Manager is responsible for registering ENS subdomains (e.g. mike.argent.xyz) and assigning them to wallets.
- **ENS Resolver:** The ENS Resolver keeps links between ENS subdomains and wallet addresses and allows to resolve them in both directions.
- **Module Registry:** The Module Registry maintains a list of the registered *Module* contracts that can be used with user wallets. It also maintains a list of registered *Upgrader* contracts that a user can use to migrate the modules used with their wallet (see Section 2.2).

A second group of contracts implements the functionalities of the wallet:

- **Modules:** Different functionalities of the wallet are encapsulated in different modules. In general, a single module contract (e.g. *Guardian-Manager*) is used by all wallets to handle a specific set of operations (e.g. adding and revoking guardians). New modules can be added, existing modules can be upgraded and old modules can be deprecated by Argent. This follows a wallet design pattern recently introduced by Nick Johnson[2]: instead of directly calling a method in their wallet to perform a given operation (e.g. transfering a token), users call a method in the appropriate module contract (e.g. *transferToken()* in the *TokenTransfer* module), which verifies that the user holds the required authorization and if so, calls an appropriate method on the wallet (e.g. *invoke()*, to call an ERC20 contract).

- **Module Storages:** Some modules store part of their states in a dedicated storage contract (see Section 2.3).

- **Proxy Wallet:** Lightweight proxy contract that delegates all calls to a Base Wallet library-like contract. There is one proxy deployed per wallet. Note that the rationale for using the Proxy-Implementation design pattern[3] is *not* to enable wallet upgradability (we use upgradable modules for that) but simply to reduce the deployment cost of each new wallet.

- **Base Wallet:** The Base Wallet is a simple library-like contract implementing basic wallet functionalities used by proxy wallets (via delegatecalls), that are not expected to ever change. These functionalities include changing the owner of the wallet, (de)authorizating modules and performing (value-carrying) internal transactions to third-party contracts.

2.2 Upgradability

Argent maintains an evolving set of registered *Module* contracts as well as a list of registered *Upgrader* contracts. *Upgrader* contracts define a migration

[2]https://gist.github.com/Arachnid/a619d31f6d32757a4328a428286da186 and https://gist.github.com/Arachnid/6a5c8ff96869fbdf0736a3a7be91b84e

[3]introduced by Nick Johnson in https://gist.github.com/Arachnid/4ca9da48d51e23e5cfe0f0e14dd6318f

from a particular set of old modules to disable to a set of new registered modules to enable. A user can perform an upgrade of their modules using one of the registered *Upgrader* contracts.

2.3 Storage

In general, each module stores the entire state pertaining to all the wallets that use that module. For example, the *TokenTransfer* module stores how much of their daily allowance has been used by each wallet. Some modules such as *TokenTransfer* make use of an additional storage contract (e.g. *TransferStorage*). This is the case when their storage needs to be accessed by other modules and/or to simplify the upgradability of the module.

2.4 Modules

2.4.1 GuardianManager module

This module is used by the wallet owner to add or revoke a guardian. The addition or revokation of a guardian is done in two steps: an addition (or revokation) step that takes 24h to complete, followed by a confirmation (or cancellation) step that needs to be done in a subsequent 12h window period.

2.4.2 LockManager module

This module is used by guardians to lock or unlock a wallet.

2.4.3 RecoveryManager module

This module is used by guardians to perform a recovery of the wallet, i.e. change the owner to a new owner. The recovery is triggered by a majority of guardians and needs to be manually finalized after a security period of 24h. Upon finalization, the RecoveryManager module calls the *setOwner()* method on the user wallet.

2.4.4 TokenTransfer module

This module lets users perform transfers of ETH and ERC20 tokens, either to whitelisted addresses without any limit, or to non-whitelisted addresses within a certain daily allowance. If the daily limit is reached for a transfer,

the transfer is set to a pending state and will only be executable by the user after 24h.

2.4.5 ApprovedTransfer module

This module lets users perform instant transfers of ETH and ERC20 tokens to non-whitelisted addresses with the signed approval of a majority of guardians.

2.4.6 TokenExchanger module

This module lets users exchange ETH or ERC20 tokens for ETH or other ERC20 tokens using Kyber Network.

2.4.7 DappManager module

This module lets users authorize third-party Dapps to call third-party contracts or transfer ETH on behalf of the user within a fixed spending limit. The authorized Dapps interact with the same module to execute the operations they have been authorized to perform.

2.4.8 ModuleManager module

This module is used by users to authorize or deauthorize modules to control their wallets or to update their current modules. The updates can only be performed by executing a registered *Upgrader* contract to perform the migration as described in Section 2.2.

2.4.9 RelayerModule

This is an abstract contract from which all modules inherit. It implements a permissionless method *execute()* that is meant to be called by a relayer account. The relayer must pass to the *execute()* function an intention and the signature(s) of this intention by the originator(s) of that intention. As described in Section 1.11, this pattern allows ether-less accounts to perform operations on the wallet without the need to directly call the corresponding module methods to do so.

The RelayerModule delegates the implementation of the code that verifies the intention and the signature(s) to the subclass modules that implement it.

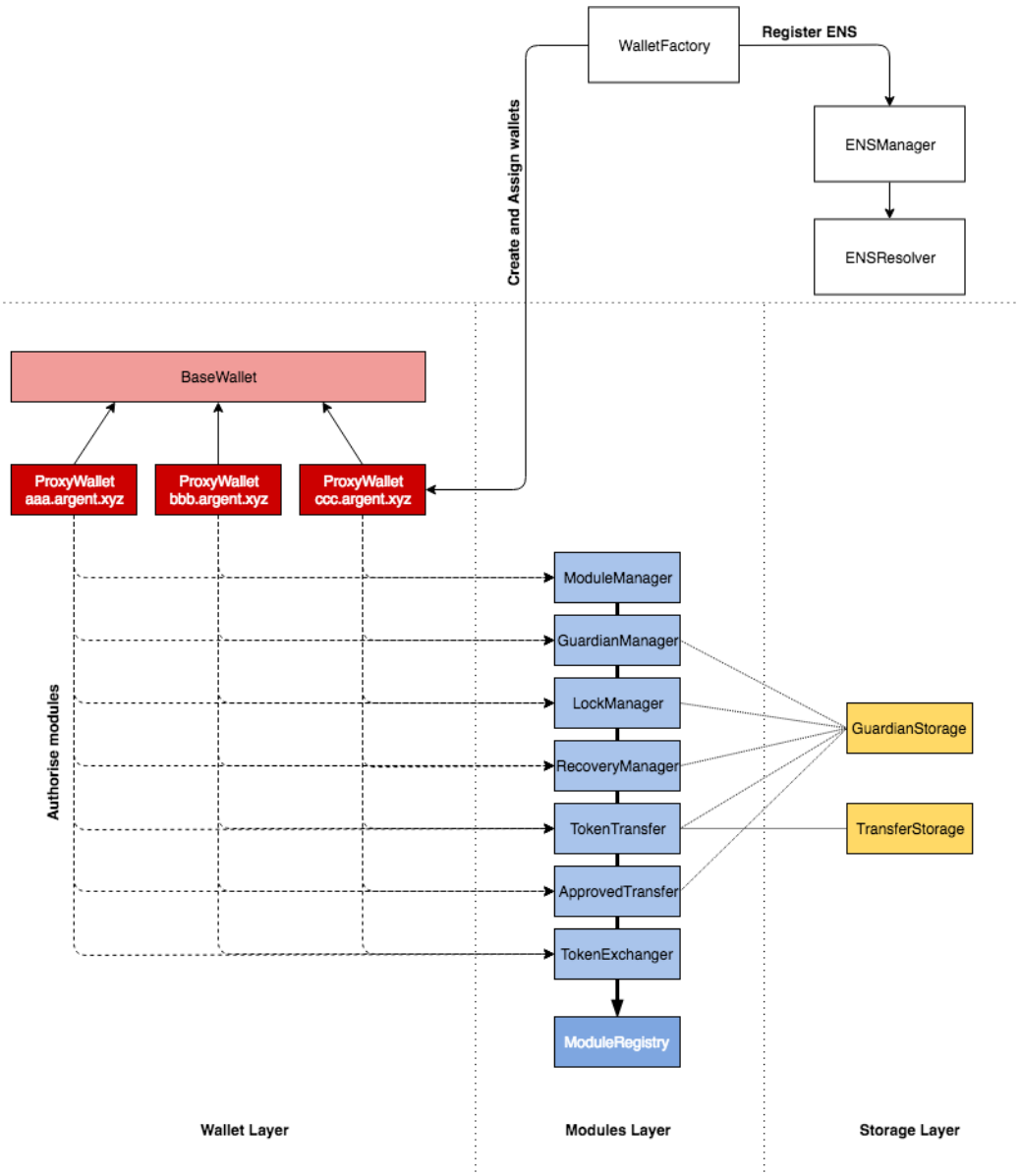

Figure 1: Smart Contracts architecture: ownership and management relationships