Gnosis Safe Documentation

Gnosis

Oct 08, 2018

Content

The Gnosis Safe aims to provide all users with a convenient, yet secure way to manage their funds and interact with decentralized applications on Ethereum. It comes in two editions: Personal Edition and Team Edition. The Gnosis Safe Personal Edition is targeting individual users using 2 or more factor authentication through native mobile apps for Android and iOS in combination with a browser extension. The Gnosis Safe Team Edition is geared towards teams managing shared crypto funds. It is an improvement of the existing Gnosis MultiSig wallet with redesigned smart contracts, cheaper setup and transaction costs as well as an enhanced user experience.

CHAPTER 1

Learn more about Gnosis Safe

- **Gnosis Safe Website**: https://safe.gnosis.io
- **The State of Storing Funds on Ethereum: Why we need the Gnosis Safe**: https://blog.gnosis.pm/the-state-of-storing-funds-on-ethereum-fdb4c9a09388
- **Announcement of Gnosis Safe Beta: Personal Edition**: https://blog.gnosis.pm/announcing-the-gnosis-safe-beta-personal-edition-19a69a4453e8

1.1 Smart Contract Overview

GitHub: https://github.com/gnosis/safe-contracts

1.1.1 Smart Contract Architecture

Gnosis Safe Transactions

A Safe transaction has the same following parameters: A destination address, an Ether value, a data payload as a bytes array, `operation` and `nonce`.

The operation type specifies if the transaction is executed as a `CALL`, `DELEGATECALL` or `CREATE` operation. While most wallet contracts only support `CALL` operations, adding `DELEGATECALL` operations allows to enhance the functionality of the wallet without updating the wallet code. As a `DELEGATECALL` is executed in the context of the wallet contract, it can potentially mutate the state of the wallet (like changing owners) and therefore should only be used with known, trusted contracts. The `CREATE` operation allows to create new contracts with bytecode sent from the wallet itself.

More information on delegate calls can be found in the solidity docs

The nonce prevents replay attacks. Each transaction should have a different nonce and once a transaction with a specific nonce has been executed it should not be possible to execute this transaction again. The concrete replay protection mechanism depends on the version of the Gnosis Safe and will be explained later.

Contract Creations

As the creation of new contracts is a very gas consuming operation, Safe contracts use a proxy pattern where only one master copy of a contract is deployed once and all its copies are deployed as minimal proxy contracts pointing to the master copy contract. This pattern also allows to update the contract functionality later on by updating the address of the master copy in the proxy contract.

As contract constructors can only be executed once at the time the master copy is deployed, constructor logic has to be moved into an additional persistent setup function, which can be called to setup all copies of the master copy. This setup function has to be implemented in a way it can only be executed once. It is important to note that the master copy contract has to be persistent and there should be no possibility to execute a `selfdestruct` call on the master copy contract.

It is **important** to know that it is possible to "hijack" a contract if the proxy creation and setup method are done in separate transactions. To avoid this it is possible to pass the initialisation data to the ProxyFactory or the Delegating-ConstructorProxy.

For more information about Proxy contracts read our blog post about Solidity DelegateProxy Contracts.

Contracts

Base Contracts

SelfAuthorized.sol

The self authorized contract implements the `authorized()` so that only the contract itself is authorized to perform actions.

Multiple contracts use the `authorized()` modifier. This modifier should be overwritten by a contract to implement the desired logic to check access to the protected methods.

Proxy.sol

The proxy contract implements only two functions: The constructor setting the address of the master copy and the fallback function forwarding all transactions sent to the proxy via a `DELEGATECALL` to the master copy and returning all data returned by the `DELEGATECALL`.

DelegateConstructorProxy.sol

This is an extension to the proxy contract that allows further initialization logic to be passed to the constructor.

PayingProxy.sol

This is an extension to the delegate constructor proxy contract that pays a specific amount to a target address after initialization.

ProxyFactory.sol

The proxy factory allows to create new proxy contracts pointing to a master copy and executing a function in the newly deployed proxy in one transaction. This additional transaction can for example execute the setup function to initialize the state of the contract.

MasterCopy.sol

The master copy contract defines the master copy field and has simple logic to change it. The master copy class should always be defined first if inherited.

EtherPaymentFallback.sol

Base contract with a fallback function to receive ether payments.

Executor.sol

The executor implements logic to execute calls, delegatecalls and create operations.

ModuleManager.sol

The module manager is an executor implementation which allows the management (add, remove) of modules. These modules can execute transactions via the module manager.

A linked list is used to store the enabled modules in the smart contract. To modify the list with minimal gas usage it is required to specify the module that should be modified and the module that points to this module. This is important when disabling a module. If multiple transactions disabling modules are submitted at once it is important to note that the module that points to the module that should be disabled might have changed. The linked list requires a sentinel (start and end pointer). This sentinel is the `0x1` address. Therefore this address cannot be used as a module.

OwnerManager.sol

The owner manager allows the management (add, remove, replace) of owners. It also specifies a threshold that can be used for all actions that require the confirmation of a specific amount of owners.

For managing the owners also a linked list (see ModuleManager.sol). Modifiying transactions that require to specify the owner pointing to the owner that should be modified include remove owner and swap owner. Also here the sentinel is the `0x1` and therefore it is not possible that this address becomes an owner.

Gnosis Safe

GnosisSafe.sol

The Gnosis Safe contract implements all basic multisignature functionality. It allows to execute Safe transactions and interact with Safe modules from internal methods. The contract provides no methods to interact with the Safe contract and also has no functionality to check if any interaction watch approved by the required amount of owners. This logic and the methods to interact with the Gnosis Safe need to be implemented by the sub-contracts.

Safe transactions can be used to configure the wallet like managing owners, updating the master copy address or whitelisting of modules. All configuration functions can only be called via transactions sent from the Safe itself. This assures that configuration changes require owner confirmations.

Before a Safe transaction can be executed, the transaction has to be confirmed by the required number of owners.

There are multiple implementations of the Gnosis Safe contract with different methods to check if a transaction has been confirmed by the required owners.

GnosisSafePersonalEdition.sol

This version is targeted at individual users, i.e. only single user would have access to all keys owning a Safe.

Once the required number of confirmations is available `execTransactionAndPaySubmitter` can be called with the sending confirmation signatures. This method will pay the submitter of the transaction for the transaction fees after the Safe transaction has been executed.

`execTransactionAndPaySubmitter` expects all confirmations sorted by owner address. This is required to easily validate no confirmation duplicates exist.

For more information check out the section about the Gnosis Safe Personal Edition.

GnosisSafeTeamEdition.sol

This version is targeted at teams where each owner is a different user. Each owner has to confirm a transaction by using `approveTransactionWithParameters`. Once the required number of owners has confirmed, the transaction can be executed via `execTransactionIfApproved`. If the sender of `execTransactionIfApproved` is an owner it is not necessary to confirm the transaction before. Furthermore this version doesn't store the nonce in the contract but for each transaction a nonce needs to be specified.

For more information check out the section about the Gnosis Safe Team Edition.

Modules

Modules allow to execute transactions from the Safe without the requirement of multiple signatures. For this Modules that have been added to a Safe can use the `execTransactionFromModule` function. Modules define their own requirements for execution. Modules need to implement their own replay protection.

Modules are smart contracts which implements a concrete Safe's functionality separating its logic from the Safe's contract. Keep in mind that modules allow the execution of transactions without needing confirmations, while this allows the implementation of many advanced use cases it also introduces additional attack vectors.

Modules can be included in the Safe according to owners' necessities, making the process of creating safes more gas efficient (not all safes should include all modules). And also open the door to developers to include its own features without compromising Safe's core functionalities, having all benefits of coding isolated smart contract. Modules that are used on a Safe should always be reviewd and audited in a similar manner as the core fuctionality of the Safe, to make sure that no exploits are introduced.

StateChannelModule.sol

This module is meant to be used with state channels. It is a module similar to the personal edition, but without the payment option (therefore the method is named `execTransaction`). Furthermore this version doesn't store the nonce in the contract but for each transaction a nonce needs to be specified.

DailyLimitModule.sol

The Daily Limit Modules allows an owner to withdraw specified amounts of specified ERC20 tokens on a daily basis without confirmation by other owners. The daily limit is reset at midnight UTC. Ether is represented with the token address 0. Daily limits can be set via Safe transactions.

SocialRecoveryModule.sol

The Social Recovery Modules allows to recover a Safe in case access to owner accounts was lost. This is done by defining a minimum of 3 friends' addresses as trusted parties. If all required friends confirm that a Safe owner should be replaced with another address, the Safe owner is replaced and access to the Safe can be restored. Every owner address can be replaced only once.

WhitelistModule.sol

The Whitelist Modules allows an owner to execute arbitrary transactions to specific addresses without confirmation by other owners. The whitelist can be maintained via Safe transactions.

Libraries

Libraries can be called from the Safe via a `DELEGATECALL`. They should not implement their own storage as this storage won't be accessible via a `DELEGATECALL`.

MultiSend.sol

This library allows to batch transactions and execute them at once. This is useful if user interactions require more than one transaction for one UI interaction like approving an amount of ERC20 tokens and calling a contract consuming those tokens. Each sub-transaction of the multi-send contract has an operation. With this it is possible to perform `CALL`s and `DELEGATECALL`s.

CreateAndAddModules.sol

This library allows to create new Safe modules and whitelist these modules for the Safe in one single transaction.

1.1.2 Safe Personal Edition Smart Contract

This version is targeted at individual users, i.e. only single user would have access to all keys owning a Safe.

Once the required number of confirmations is available `execTransactionAndPaySubmitter` can be called with the sending confirmation signatures. This method will pay the submitter of the transaction for the transaction fees after the Safe transaction has been executed.

Safe Contract Deployment

The Gnosis Safe Personal Edition smart contract was written with the usage of a proxy contract in mind. Because of that there is no constructor and it is required to call an inilize function on the contract before it can be used. For this it is recommended to use the ProxyFactory or the DelegatingConstructorProxy.

Using the ProxyFactory or deploying a proxy requires that the user has ether on an externally owned account. To make it possible to pay for the creation with any token or ether the following flow is used.

1. Create deployment transaction. The PayingProxy enables the payment in any ERC20 token. Once the proxy is deployed it will refund a predefined address with the funds present at the address where it was deployed.

2. To make the deployed address deterministic it is necessary to use a known account and calculate the target address. To make this trustless it is recommended to use a random account that has nonce 0. This can be done by creating a random signature for the deployment transaction. From that transaction it is possible to derive the sender and the target address.
3. The user needs to transfer at least the amount required for the payment to the target address.
4. Once the payment is present at the target address the relay service will fund the sender with ether required for the creation transaction.
5. As soon as the sender is funded the creation transaction can be submitted.

For more details on the Safe deployment process please checkout the DappCon 2018 presentation

Safe Transaction Execution

To execute a transaction with the Gnosis Safe Personal Editon the `execTransactionAndPaySubmitter` methods needs to be called with the following parameters:

- to, value, data - Safe transaction information
- operation - Operation that should be used for the Safe Transaction. Can be `CALL` (uint8 - `0`), `DELEGATECALL` (uint8 - `1`) or `CREATE` (uint8 - `2`)
- safeTxGas - minimum gas provided for the Safe transaction. In case of `CALL` and `DELEGATECALL` this is also the maximum available gas (gas limit).
- dataGas - overhead gas to execute the Safe transaction
- gasPrice - price used to calculate the gas costs that are refunded to the submitter.
- gasToken - Token used for gas cost payment. If `0x0` then Ether is used. Gas costs are calculated by `(dataGas + txGas) * gasPrice`
- signatures - hex encoded signatures (`execTransactionAndPaySubmitter` expects that the signatures are sorted by owner address. This is required to easily validate no confirmation duplicates exist)

There need to be enough signatures to reach the threshold configured on Safe setup. To generate a signature a Safe owner signs the keccak hash of the following information: `byte(0x19), byte(0), this, to, value, data, operation, safeTxGas, dataGas, gasPrice, gasToken, _nonce` where `this` is the Safe address and `_nonce` is the the global `nonce` stored in the Safe contract.

The `nonce` of the Safe contract is a public variable and increases after each execution of a Safe transaction (every time `execTransactionAndPaySubmitter` is executed).

When a transaction was submitted the contract will store the gas left on method entry. Based on this the contract will calculate the gas used that the user needs to pay.

Before executing the Safe transaction the contract will check the signatures of the Safe, to ensure that the transaction was authorized by the Safe owners, and check that enough gas is left to fullfill the gas requested for the Safe transaction (`safeTxGas`). If these checks fail the transaction triggering `execTransactionAndPaySubmitter` will also fail. This means the submitter will not be refunded.

After the execution of the Safe transaction the contract calculates the gas that has been used based on the start gas. If the `gasPrice` is set to 0 no refund transaction will be triggered.

Refunds are not included in the calculated gas costs, since the contract uses `gasLeft()` to calculate how much gas has been used.

Failing Safe Transactions

If the execution of a Safe transaction fails the contract will emit the `ExecutionFailed` event that contains the transaction hash of the failed transaction. The transaction triggering `execTransactionAndPaySubmitter` will not fail, since the submitter should still be refunded in this case.

Safe Transaction Gas Limit Estimation

The user should set an appropriate `safeTxGas` to define the gas required by the Safe transaction, to make sure that enough gas is send by the submitter with the transaction triggering `execTransactionAndPaySubmitter`. For this it is necessary to estimate the gas costs of the Safe transaction.

To correctly estimate the call to `execTransactionAndPaySubmitter` it is required to generate valid signatures for a successful execution of this method. This opens up potential exploits since the user might have to sign a very high `safeTxGas` just for estimation, but the signatures used for the estimation could be used to actually execute the transaction.

One way to estimate Safe transaction is to use `estimateGas` and with the following parameters:

```
{
    "from": <Safe address>,
    "to": <`to` of the Safe transaction>,
    "value": <`value` of the Safe transaction>,
    "data": <`data` of the Safe transaction>,
}
```

While it is possible to estimate a normal transactions (where `operation` is `CALL` or `CREATE`) like this, it is not possible to estimate `DELEGATECALL` transactions this way. Also the value returned by `estimateGas` includes refunds and the base transaction costs.

For a more accurate estimate it is recommended to use the `requiredTxGas` method of the Safe contract. The method takes `to`, `value`, `data` and `operation` as parameters to calculate the gas used in the same way as `execTransactionAndPaySubmitter`. Therefore it will not include any refunds.

To avoid that this method can be used inside a transaction two security measures have been put in place:

1. The method can only be called from the Safe itself
2. The response is returned with a revert

The value returned by `requiredTxGas` is encoded in a revert error message (see solidity docs at the very bottom). For retrieving the hex encoded uint value the first 68 bytes of the error message need to be removed.

Safe Transaction Data Gas Estimation

The `dataGas` parameter can be used to include additional gas costs in the refund. This could include the base transaction fee of 21000 gas for a normal transaction, the gas for the data payload send to the Safe contract and the gas costs for the refund itself.

To correctly estimate the gas costs for the data payload without knowing the signatures, it is suggested to generate the transaction data of `execTransactionAndPaySubmitter` with random signatures and `dataGas` set to 0. The costs for this data are 4 gas for each zero-byte and 68 gas for each non-zero-byte.

Signature Encoding

Assuming we have 2 owners in a 2 out of 2 multisig configuration:

1. `0x1` (Private key)
2. `0x2` (Private key)

`0x1` and `0x2` are confirming by signing a message.

The signatures bytes used for `execTransactionAndPaySubmitter` have to be build like the following:

- `bytes = 0x{r_0x1}{s_0x1}{v_0x1}{r_0x2}{s_0x2}{v_0x2}`

`v`, `r` and `s` are the signature parameters for the signed confirmation messages. All values are hex encoded. `r` and `s` are padded to 32 bytes and `v` is padded to 8 bytes.

Front Running Submitted Transactions

As all the parameters required for execution are part of the submitted transaction it is possible that miners front-run the original submitter to receive the reward. In the long run that behaviour would be appreciated, since it would allow that anybody submits these transactions with `gasPrice` of the transaction triggering `execTransactionAndPaySubmitter` set to 0. Miners could pick up these transactions and claim the rewards.

As it might be undesirable when there is a specific relayer (submitter) that the owner of the Safe wants to interact with it is currently in discussion to allow the specification of the address that should be refunded and fallback to the original behaviour if that address is `0x0` issue link

1.1.3 Safe Team Edition Smart Contract

This version is targeted at teams where each owner is a different user. Each owner has to confirm a transaction by using `approveTransactionWithParameters`. Once the required number of owners has confirmed, the transaction can be executed via `execTransactionIfApproved`. If the sender of `execTransactionIfApproved` is an owner it is not necessary to confirm the transaction before. Furthermore this version doesn't store the nonce in the contract but for each transaction a nonce needs to be specified.

Failing Safe Transactions

If the execution of a Safe transaction fails then the transaction triggering `execTransactionIfApproved` will also fail. The existing approvals will not be reset. Therefore it is possible that the transaction at a later point can be retried without the requirement that everybody approves the transaction again. This is important to consider when owners of a Safe change, since it might be possible that at a later point in time the Safe transaction doesn't fail anymore, thus allowing execution.

1.2 Services Overview

1.2.1 Notification Service

Allows users to send signed transaction messages between devices taking part in the signing process.

Show on GitHub

1.2.2 Relay Service

This service allows us to have owners of the Safe contract that don't need to hold any ETH on those owner addresses, why? Because the transaction relay service acts as a proxy, paying for the transaction fees and getting it back due to the transaction architecture we use.

Show on GitHub

1.2.3 History Service

Keeps track of transactions sent via Gnosis Safe contracts and confirmed transactions.

Show on GitHub

1.3 Safe Personal Edition Clients

1.3.1 Android Client for the Safe Personal Edition

Show on GitHub

1.3.2 iOS Client for the Safe Personal Edition

Show on GitHub

1.3.3 Chrome Extension for the Safe Personal Edition

Show on GitHub

1.4 Safe Team Edition Clients

1.4.1 Web React Client for the Safe Team Edition

Show on GitHub