# StormX Token Swap Documentation

## Overview



## Executive Summary

StormX currently operates an ERC20 token deployed on Ethereum mainnet address `0xd0a4b8946cb52f0661273bfbc6fd0e0c75fc6433`. The token contract includes features for privileged access that allow StormX to mint new tokens for and remove tokens from arbitrary accounts. The StormX team sought to develop a new ERC20 token smart contract that will not include the aforementioned functions. Additionally, the token should include a new feature for locking tokens (so-called “staking”). The StormX team sought to reward users who choose to hold and lock their tokens with tokens from their own reserves.

This document provides information about the developed solution.

## Assumptions

| # | Assumption |
|---|---|
| A1 | The new token will become the owner of the original token smart contract so that it can call the owner-only functions in the original token smart contract. |
| A2 | After tokens are locked, users alway have to manually unlock the locked tokens if they want to manipulate these tokens. |

| | |
|---|---|
| A3 | The privileged access (function `destroy()`) can be used for migration. |
| A4 | StormX will embed support for staking and unstaking into their Chrome plugin. |

## Non-requirements

| # | Non-requirement |
|---|---|
| N1 | StormX will have an off-chain job sending tokens accumulated as interest to users' accounts based on the locked balance. The implementation will not support reward beyond reading the locked token balance of users. |
| N2 | After migration is closed, existing StormX tokens will not be destroyed for users who do not wish to migrate. |
| N3 | The token swap website does not need to provide functionality for staking and unstaking. |

## Requirements

The following requirements were gathered and agreed upon before the development:

### Standard ERC20 interface

| # | Requirement |
|---|---|
| R1-1 | Supports the standard ERC20 interface and include methods. |
| R1-2 | Supports two standard ERC20 events `Transfer` and `Approval` . |
| R1-3 | The `name` of the new token is “StormX“. |
| R1-4 | The `symbol` of the new token is “STMX“. |
| R1-5 | The `decimals` of the new token remains the same. |
| R1-6 | The new token supports a function `transfers()`for batch transfer of tokens. |
| R1-7 | The signature of function `transfers()` is identical with the signature of the function `transfers()` in the original token. |
| R1-8 | The function `transfers()` is available to everyone. |
| R1-9 | The function `transfers()` is available only when transfers is not disabled. |
| R1-10 | The function `enableTransfers()` is only available to StormX. |
| R1-11 | The new token contract is ownable. |
| R1-12 | StormX is the owner of the new token contract. |

### MetaTransaction supported via GSN

| # | Requirement |
|---|---|
| R2-1 | The new token smart contract is GSN-capable and is able to receive transactions from GSN. |
| R2-2 | Each accepted meta transactions via GSN charges the user a certain amount of StormX tokens. The default charge fee is 10 StormX (unless changed by StormX; see R2-3). |
| R2-3 | The fee for GSN network is settable by StormX. |
| R2-4 | Fees charged for transactions relayed by GNS are sent to an Ethereum address. |
| R2-5 | StormX can change the address for collecting fees for GSN at any point. |
| R2-6 | Users can send meta transactions via GSN and the relayed call should be accepted as long as the users have enough token balance. |

| | |
|---|---|
| R2-7 | The new token smart contract also supports receiving transaction directly. |
| R2-8 | The meta transaction call via GSN is rejected with the error code `NO_ENOUGH_BALANCE` if the user does not have enough balance. |

### Security Emphasis

| # | Requirement |
|---|---|
| R3-1 | The token contract includes methods `increaseAllowance()` and `decreaseAllowance()` to mitigate allowance double-spend exploit |
| R3-2 | The token is not pausable. |
| R3-3 | The privileged access in the form of functions `destroy()` and `issue()` is removed. |

### Token migration

| # | Requirement |
|---|---|
| R4-1 | The token swap contract allows for a token swap from old StormX token to the new one. |
| R4-2 | The conversion guarantees that the total supply of the new token match exactly with that of the original token. |
| R4-3 | The swap is possible for at least 6 months. Then, StormX can stop it. After stopping the swap, the rest of the tokens should be minted and sent to StormX's reserve, so the total supply will reach the original StormX total supply. |
| R4-4 | Users do not need to provide approvals for the tokens to migrate. |

### Staking feature

| # | Requirement |
|---|---|
| R5-1 | Users can lock and unlock their tokens |
| R5-2 | Users cannot manipulate locked tokens by any means. |
| R5-3 | Locked tokens are still reported as owned by the user when method `balanceOf()` is called. |
| R5-4 | Users can transfer their tokens after unlocking them. |
| R5-5 | Anyone can read the locked token balance of a user. |

### Token Swap Website Requirements

| # | Requirement |
|---|---|
| R6-1 | The website is to be hosted on Heroku as its own app with (sub-)domain swap.stormx.io. |
| R6-2 | The website must allow for swapping old StormX token to the new StormX token. |
| R6-3 | The visual look of the website must respect branding guide of StormX with UX similar to kyberswap.com. |
| R6-4 | The website must support TrustWallet. |
| R6-5 | The website must support Ledger. |
| R6-6 | The website must support interaction by uploading keystore, entering private key, or mnemonic. |
| R6-7 | The website must support submitting transactions to GSN network. |
| R6-8 | The website must support submitting transactions directly to the swap smart contract. |
| R6-9 | The website must be covered by unit and e2e tests. |
| R6-10 | The website should visualize if the swap phase is still open or not (see R4-3 for context). |

*(See also the N3 for the website non-requirements.)*

## Technical Executions

Quantstamp developed the contracts according to the requirements using Solidity and Javascript for testing. This section outline the technical solution.

### StormXGSNRecipient

This contract is an ownable contract for supporting GSN relayed calls and charging users for accepted GSN calls. Both `StormXToken` and `Swap` inherits from this contract. For every accepted GSN relayed call, the user will be charged for a specified amount of StormXTokens, i.e.`chargeFee`, and the charged tokens will be sent to `stormXReserve`. If the user does not have enough unlocked token balance, the GSN relayed call will be rejected and the user will not be charged at all.

Only the contract owner can call the methods `setChargeFee(uint256 newFee)` and`setStormXReserve(address newReserve)` to set `chargeFee` and `stormXReserve` respectively.

### StormXToken

StormXToken is the new token contract implemented for StormX. It supports standard ERC20 interface, transferring in batch, staking feature and GSN relayed calls.

#### Standard ERC20 interface

StormXToken is in compliance with ERC20 as described in eip-20.md. This token contract is ownable and mintable. Caller of the constructor will become the owner and only the owner will be able to add minters for this token contract. For security concern, minters should not mint any tokens after token migration is closed (see Security Concern S-1).

#### Transferring in batch

Anyone can call the method `transfers()` to perform multiple transferring in a single function call.

```
function transfers(
    address[] memory recipients,
    uint256[] memory values
  ) public transfersAllowed returns (bool)
```

Only contract owner can enable/disable the method `transfers()` by invoking the method `function enableTransfers(bool enable) public onlyOwner`. It enables method `transfers()`for batch transfers if `enable=true`, and disables `transfers()` otherwise.

#### Staking Feature

The new token will include a staking feature and StormX will reward users for any staked tokens they have.

This feature comprises the following sections.

### Lock

By invoking the function `lock()`, users will be able to lock any amount of new StormX tokens as long as they have enough token balance. Locked tokens will not be manipulable by any means. Once the specified amount of tokens are locked successfully, an interest start to be accumulated and calculated off-chain by StormX (see N1). The balance of locked tokens users will be readable via read methods (see section Read Methods). While the users will not be able to perform any operations on locked tokens, these locked tokens will still be reported as owned by the users when method `balanceOf()` is called.

### Unlock

By invoking the function `unlock()`, users will be able to unlock any amount of locked new StormX tokens they have, and will be able to perform any operations on their unlocked tokens as desired. Once the specified amount of tokens becomes unlocked, those tokens will no longer accumulate interest.

## Read Methods

Anyone can call read methods to retrieve the different kinds of balance of an account.

1. `lockedBalanceOf(address account)` returns the amount of locked tokens `account` holds
2. `unlockedBalanceOf(address account)` returns the amount of unlock tokens `account` holds
3. `balanceOf(address account)` returns the total amount of tokens `account` holds, i.e the sum of locked and unlocked tokens

#### GSN relayed calls

StormXToken contract inherits from `StormXGSNRecipient.sol` (see StormXGSNRecipient section for more details) and is able to receive GSN relayed calls from GSN relay hub. For references, see [1,2].

The function `acceptRelayedCall` implemented in `StormXGSNRecipient.sol` decides whether to accept a relayed call from GSN or not. As per current implementation, if the user has no less than specified charge fee `chargeFee`, the relayed call will be accepted and the user will be charged the specified fee.

Only the contract owner can call the methods `addGSNRecipient(address recipient)` and `deleteGSNRecipient(address recipient)` to add and delete GSNRecipient respectively. For any contract inheriting from `StormXGSNRecipient` that will charge users, it must be added as GSNRecipient in this token contract, otherwise the charging will fail. If a valid GSNRecipient is deleted by the contract owner, it will also fail to charge users for any fees.

Exception: If a contract inherits from `StormXGSNRecipient` and is deployed at address `recipient`, if the user invoke `StormXToken.approve(recipient, chargeFee)` before the GSN relayed call, this contract can successfully charge the user for a GSN relayed call.

### Swap

This smart contract supports the token migration and guarantees that the total supply of the new token will be at most the total supply of the original token (the token supply will in fact be equal to the original token supply; see R4-3)

#### Token Migration

To open token migration, ownership of the old token contract should be transferred to this contract. The contract owner should call the function `initialize()` to accept the old token contract ownership and record the initialized time to guarantee the token swap can last at least 24 weeks.

Anyone can call the method `convert(uint256 amount)` to convert a specified amount of original StormX tokens to the new token with exchange rate 1:1 as long as the user has enough unlocked token balance.

The migration uses the privileged access of the original token to dispose of the amount being migrated, and token minting for the new StormX token. Therefore, the user will not need to issue approvals for the tokens undergoing migration.

The new token will be compliant with the principles of decentralization, i.e., it will be the choice of a user to convert their original tokens to the new tokens. StormX will not be able to force or deny such a conversion for any account.

#### Close Token Migration

The token migration can be stopped by StormX only after 24 weeks from initialization. Only the contract owner can call the function `disableMigration(address reserve)` to stop token swap, which will mint remaining tokens and send them to the address `reserve`.

#### Transfer Ownership

Only the contract owner can call the method `transferOldTokenOwnership(address newOwner)` to transfer the old token contract ownership to `newOwner`, and `newOwner` have to accept the ownership explicitly by calling the function `acceptOwnership()` in old token contract to accept the ownership.

# Deployment Instructions

The following order is strictly required when deploying relevant contracts and setting up initializations

1. StormXAdmin deploys `StormXToken`, passing the address of StormX's reserve. StormXAdmin becomes the owner of the contract and StormX's reserve will receive all charged fees of GSN relayed calls
    a. verify that StormXAdmin is the owner of `StormXToken` contract
2. StormXAdmin deploys `Swap`, passing the address of `StormToken`(the old token contract), `StormXToken` and StormX's reserve. StormXAdmin becomes the owner of `Swap` and StormX's reserve will receive all charged fees of GSN relayed calls
3. StormXAdmin invokes the function `StormXToken.addMinter(Swap.address)` so that `Swap`can mint new tokens during token swap
4. StormXAdmin invokes the function `StormXToken.addGSNRecipient(Swap.address)`so that `Swap`can charge users for GSN relayed calls
5. StormXAdmin invokes the functions `StormToken.transferOwnership(Swap.address)` and `Swap.initialize()` so that `Swap`can destroy old tokens for users during token swap

## Non-Trivial Use-Cases

### UC1.1 User converts original tokens to new StormX token

1. The user calls the function `convert(amount)` with signed signature to GSN.
2. The new token smart contract accepts the relayed call from GSN and execute `convert(amount)`.
3. The function checks whether the `_msgSender()`(i.e the original caller) has this amount of original StormX token, and reverts if not.
4. The function calls the function `destroys(userAddress, amount)` of original StormX tokens from the original caller (this requires the new token becomes the owner of the original token smart contract; see A1).
5. The function increases total supply of new StormX token by the specified `amount`.
6. The function increases balance of the `_msgSender()` by the specified `amount`.
7. The event `TokenConverted(userAddress, amount)` is emitted to indicate the success of conversion.
8. The function returns `true`.

#### UC1.2 User locks some amount of new StormX tokens and earn interests

1. The user calls the function `lock(amount)` with signed signature to GSN.
2. The new token smart contract accepts the relayed call from GSN and execute `lock(amount)`.
3. The function checks whether the `_msgSender()`(i.e the original caller) has this amount of new StormX token, and reverts if not.
4. The function increases `lockedBalance` of the original caller by `amount`.
5. The function decreases balance of the original token by `amount`.
6. The function emits the event `TokenLocked(userAddress, amount)` to indicate the success of staking.
7. The function returns `true` and interests earn from these newly-locked tokens starts to accumulate.

#### UC1.3 User unlocks some amount of new StormX tokens and view the balance of lockedBalance

1. The user calls the function `unlock(amount)` with signed signature to GSN.
2. The new token smart contract accepts the relayed call from GSN and execute `unlock(amount)`.
3. The function checks whether the `_msgSender()`(i.e the original caller) has this amount locked StormX token, and reverts if not (or possibly, unlock as much as the user has; clarification is needed if this is preferred).
4. The function decreases `lockedBalance` of the original caller by `amount`.
5. The function increases balance of the original token by `amount`.
6. The function emits the event `TokenUnlocked(userAddress, amount)` to indicate the success of unstaking.
7. The function returns `true` .
8. The user calls the function `lockedBalanceOf(address)`.
9. The new token smart contract executes `lockedBalanceOf()`.
10. The function returns `lockedBalance[userAddress]`.

#### UC1.4 StormX admin closes token migration and collects all the rest of original tokens

1. StormX admin calls the function `disableMigration()`.
2. This function checks the timestamp and returns `false` immediately if it has not been 6 months since the construction of the new token smart contract.
3. If it has already been 6 months, this function sets `migrationOpen` to `false`, emits the event `MigrationClosed()` .

4. This function calls `mintAndTransferAsset()` and the event `MigrationLeftoverTransferred(stormX, amount)` is emitted.
5. This function returns `true`.

#### UC1.5 The user sends a meta transaction to GSN and this token contract accepts the relayed call from GSN

1. User sends a function with signed data to GSN relayHub.
2. `acceptRelayedCall()` in the new token contract is called by some relayer in GSN relayHub

```
function acceptRelayedCall(
        address relay,
        address from,
        bytes calldata encodedFunction,
        uint256 transactionFee,
        uint256 gasPrice,
        uint256 gasLimit,
        uint256 nonce,
        bytes calldata approvalData,
        uint256 maxPossibleCharge
    )
        external
        view
        returns (uint256, bytes memory);
```

3. The new token contract then checks whether `balance[from] >= chargeFee` .
4. Accepts meta transactions originated by the user `from` if the above returns true, transfers the user's balance with amount `chargeFee` to `stormXReserve` and returns `_approveRelayedCall()`.
5. Otherwise, `acceptRelayedCall()` returns `_rejectRelayedCall(uint256 errorCode)` which indicates the failure and rejects the relayed call, with the `errorCode` being `NO_ENOUGH_BALANCE`.

#### UC1.6 Send transactions via GSN

Note: more details regarding user instructions will be added after the implementation is done.

1. The user signs the data {`relayerAddress`, `userAddress`, `GSNRecipientAddress`, `encodedFunction`, `transactionFee`, `gasPrice`, `gasLimit` ,`nonce`, `relayHubAddress`}
    a. `relayerAddress` Address of the relayer that the user wants to send request to. The relayer must have already been registered in GSN for the request to succeed
    b. `userAddress` Address of the user who signs the data and wants to send the meta transaction via GSN
    c. `GSNRecipientAddress` The target contract, i.e. the address of the new token contract
    d. `encodedFunction` The function call to relay
    e. `transactionFee` The fee(%) the relay takes over the actual gas cost
    f. `gasPrice` The gas price that the user is willing to pay
    g. `gasLimit` The limit the user wants to put on the transaction
    h. `nonce` The target relayer's nonce for avoiding replay attack
    i. `relayHubAddress` Address of the relayHub
2. The user sends the data and the signature to a registered relayer
3. The registered relayer sends the transaction for the user.

## References


1. Interacting with RelayHub: https://docs.openzeppelin.com/gsn-provider/0.1/interacting-with-relayhub
2. Source code of RelayHub: https://etherscan.io/address/0xD216153c06E857cD7f72665E0aF1d7D82172F494#code
3. ERC20: eip-20.md

4. GSN: https://docs.openzeppelin.com/contracts/2.x/api/gsn
5. To write a GSN-callable contract: https://docs.openzeppelin.com/contracts/2.x/gsn
6. Source code of the original StormX token: https://etherscan.io/address/0xd0a4b8946cb52f0661273bfbc6fd0e0c75fc6433#code