certora

Security Assessment & Formal Verification Report

tether

TetherToken

Feb 2024

Prepared for
Tether

Table of content

Project Summary

Project Scope

Repo Name	Repository	Last commit	Compiler version	Platform
tether-contracts-hardhat	https://github.com/tetherto/tether-contracts-hardhat	0eea2d3	0.8.4	EVM

Project Overview

This document describes the specification and verification of the TetherToken using the Certora Prover and manual code review findings. The work was undertaken from **19 February 2024** to **29 February 2024**.

The following contract list is included in our scope:

contracts/Tether/TetherToken.sol

contracts/Wrappers/CeloExtension.sol

contracts/Wrappers/FeeCurrencyWrapper.sol

contracts/Wrappers/WrappedExtension.sol

The Certora Prover demonstrated the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts. During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed below.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Acknowledged	Code Fixed
Critical	-	-	-
High	1	1	1
Medium	1	1	1
Low	2	2	2
Informational	5	-	-
Total	9	4	4

Detailed Findings

High Severity Issues

H-1. A blocked user can burn their funds to retrieve the original token before the `owner` could call `destroyBlockedFunds()`, bypassing the blocklist mechanism

Description

In the `TetherToken` contract, the operations on funds have a blocklist preventing blocked users from transferring their funds.

In the `WrappedExtension is TetherToken` contract, there's a `WrappedExtension.withdraw()` function which allows any user to burn their funds and get back the same amount of `originalToken` in exchange.

A blocked user is a user on which `TetherToken.destroyBlockedFunds()` should be called. However, this blocked user has the possibility to frontrun the admin's destruction operation with a call to `WrappedExtension.withdraw()` to exchange their blocked funds against original tokens, allowing them the freedom to move them elsewhere afterwards, effectively circumventing the blocklist.

Recommendation

The line of code `require(!isBlocked[from], "Sender is blocked")` is copy-pasted multiple times in the codebase, which is error-prone.
One global solution could be, in `TetherToken`, to override the `_beforeTokenTransfer()` function that gets called before any transfer, mint or burn operation, to check against the blocklist. It should be noted that there'll be a need to unblock and re-block the blocked user in `TetherToken.destroyBlockedFunds()` to be able to destroy the funds due to the call of `_beforeTokenTransfer()` in `_burn()` at TetherToken.sol#L223 which would revert if the user is blocked. The added benefit is that this solution would erase any need to remember applying the check in future iterations of the code.

Customer's response: changed

Medium Severity Issues

M-1. Breaking the property of "Funds cannot be transferred to TetherToken"

Description:

In `TetherToken.transferWithAuthorization()`, the checks for `msg.sender` not being blocked and `from` not being blocked are made.
However, compared to `TetherToken.transferFrom()`, the check for `_recipient != address(this)` is missing, and then `_transfer()` is directly called.
As nothing prevents `to == address(this)`, this means that it's possible to circumvent the property of "Funds cannot be transferred to TetherToken" that is attempted to be enforced in the protocol and actually transfer funds to the TetherToken contract.

Customer's response: changed in beforeTokenTransfer

Low Severity Issues

L-1. Do not leave an implementation contract uninitialized

Description:

An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy.
To prevent the implementation contract from being used, it's advisable to initialize it in the constructor to automatically lock it when it is deployed.
One way to do it is to add the initializer modifier to the constructor:

- WrappedExtension.sol#L33-L35

```
Unset
File: WrappedExtension.sol
- 33:     constructor(address _originalToken) {
+ 33:     constructor(address _originalToken) initializer {
34:         originalToken = _originalToken;
35:     }
```

Sources:

- https://docs.openzeppelin.com/contracts/4.x/api/proxy#Initializable-_disableInitializers--
- https://twitter.com/0xCygaar/status/1621417995905167360?s=20

Customer's response: added initializer modifier to TetherToken constructor

L-2. Use the Namespaced Storage pattern instead of `TetherToken`'s `isTrusted` for upgradable contracts missing a `_gap`

Description:

TetherToken's isTrusted is used as a placeholder to preserve storage slots across upgrades:

```
Unset
File: TetherToken.sol
30:   // Unused variable retained to preserve storage slots across upgrades
31:   mapping(address => bool) public isTrusted;
```

While it could be argued that it would be better to add a `__gap` storage variable to allow for new storage variables in later versions (OZ docs), there's also the newest and most adopted pattern of "Namespaced Storage".
This is used in all of the newest OpenZeppelin dependencies.
This video at this timestamp explains it well.

As the following upgradeable contracts don't have `_gap` variables, consider refactoring the code to use "Namespaced Storage":

Affected code:

- contracts/Tether/TetherToken.sol
- contracts/Wrappers/FeeCurrencyWrapper.sol
- contracts/Wrappers/WrappedExtension.sol
- contracts/Wrappers/ArbitrumExtension.sol
- contracts/Wrappers/CeloExtension.sol

Customer's response: Added Gaps to TetherToken, EIP3009 and WithBlockedList

Informational Severity Issues

INFO-1 `require(!isBlocked)` statements are returning an erroneous error string

The revert string is returned when the require statement doesn't pass and the reason for the error needs to be returned. For the check against blocklisted users, the revert string should be `Sender is blocked` instead of `Sender is not blocked` (or instead of nothing):

Unset

```
- TetherToken.sol:193:          require(!isBlocked[_sender]);
+ TetherToken.sol:193:          require(!isBlocked[_sender], "Sender is blocked");

- TetherToken.sol:274:          require(!isBlocked[from], "Sender is not blocked");
+ TetherToken.sol:274:          require(!isBlocked[from], "Sender is blocked");

- TetherToken.sol:331:          require(!isBlocked[from], "Sender is not blocked");
+ TetherToken.sol:331:          require(!isBlocked[from], "Sender is blocked");

- TetherToken.sol:393:          require(!isBlocked[from], "Sender is not blocked");
+ TetherToken.sol:393:          require(!isBlocked[from], "Sender is blocked");

- TetherToken.sol:452:          require(!isBlocked[from], "Sender is not blocked");
+ TetherToken.sol:452:          require(!isBlocked[from], "Sender is blocked");
```

INFO-2 Draft Dependencies

Draft contracts have not received adequate security auditing or are liable to change with future developments.

Affected code:

- contracts/Tether/TetherToken.sol

```
Unset
# File: contracts/Tether/TetherToken.sol

TetherToken.sol:7:                                                                import
"@openzeppelin/contracts-upgradeable/token/ERC20/extensions/draft-ERC20PermitUpgradeable.sol"
;
```

INFO-3 Duplicated `require()`/`revert()` Checks Should Be Refactored To A Modifier Or Function

Affected code:

- contracts/Tether/TetherToken.sol

```
Unset
contracts/Tether/TetherToken.sol:
  177:         require(
  178             _recipient != address(this),
  179             "ERC20: transfer to the contract address"
  180         );

  189:         require(
  190             _recipient != address(this),
  191             "ERC20: transfer to the contract address"
  192         );
```

```
Unset
TetherToken.sol:193:        require(!isBlocked[_sender]);
TetherToken.sol:274:        require(!isBlocked[from], "Sender is not blocked");
TetherToken.sol:331:        require(!isBlocked[from], "Sender is not blocked");
TetherToken.sol:393:        require(!isBlocked[from], "Sender is not blocked");
TetherToken.sol:452:        require(!isBlocked[from], "Sender is not blocked");
```

INFO-4 `require()` statements should have descriptive reason strings

Affected code:

- contracts/Tether/TetherToken.sol

```
Unset
# File: contracts/Tether/TetherToken.sol

- TetherToken.sol:193:        require(!isBlocked[_sender]);
+ TetherToken.sol:193:        require(!isBlocked[_sender], "Sender is blocked");

- TetherToken.sol:221:        require(isBlocked[_blockedUser]);
+ TetherToken.sol:221:        require(isBlocked[_blockedUser], "Sender is not blocked");
```

- contracts/Wrappers/WrappedExtension.sol

```
Unset
# File: contracts/Wrappers/WrappedExtension.sol

- WrappedExtension.sol:59:        require(canWithdraw);
+ WrappedExtension.sol:59:        require(canWithdraw, "Withdrawals are disabled");

- WrappedExtension.sol:66:        require(_differentToken != originalToken);
+ WrappedExtension.sol:66:        require(_differentToken != originalToken, "input token is
originalToken");
```

INFO-5 addresses shouldn't be hardcoded

It is often better to declare addresses as immutable, and assign them via constructor arguments. This allows the code to remain the same across deployments on different networks and avoids recompilation when addresses need to change.

Affected code:

- contracts/Wrappers/CeloExtension.sol

```
Unset
# File: contracts/Wrappers/CeloExtension.sol

CeloExtension.sol:24:   address constant private GAS_FEE_ADDRESS =
0x000000000000000000000000000000000Ce106A5;
```

This is very specific to this implementation

INFO-6 blocked user in a wrapped token can send funds wrapper

If by mistake originalToken Owner is the wrapped contract, a user blocked in the originalToken will be able to send its blocked funds to the wrapped and get “clean” new wrapped tokens.

Affected code:

- contracts/Wrappers/WrappedExtension.sol

```
Unset
# File: contracts/Wrappers/WrappedExtension.sol

52: function deposit(uint _value) public returns (bool success) {
53        IERC20Upgradeable(originalToken).safeTransferFrom(msg.sender, address(this),
_value);
54        _mint(msg.sender, _value);
55       return true;
56     }
```

INFO-7 WrappedExtension.deposit will be denied of service when zero address is blocked

If by mistake address zero is blocked, the deposit function will be out of service.

Affected code:

- contracts/Wrappers/WrappedExtension.sol

```
Unset
# File: contracts/Wrappers/WrappedExtension.sol

52: function deposit(uint _value) public returns (bool success) {
53        IERC20Upgradeable(originalToken).safeTransferFrom(msg.sender, address(this),
_value);
54        _mint(msg.sender, _value);
55       return true;
56     }
```

INFO-8 possible re-entrance option in FeeCurrencyWrapper.creditGasFees when wrapping CeloExtension

Re-entrancy safety meaning that all storage accesses occur either before external call or after. The external call is for wrappedToken.creditGasFees. It should be fine as long as wrappedToken=CeloExtension as there is trust between FeeCurrencyWrapper and CeloExtension.

Affected code:

- contracts/Wrappers/FeeCurrencyWrapper.sol

```
Unset
# File: contracts/Wrappers/FeeCurrencyWrapper.sol
90-132: function creditGasFees() external onlyVm {...}
```

Gas Optimizations Recommendations

G-1. `TetherToken` doesn't need to directly inherit from `Initializable` and `OwnableUpgradeable`

Initializable and OwnableUpgradeable are already inherited through other parent contracts:

Unset

```
File: draft-ERC20PermitUpgradeable.sol
22:   abstract   contract   ERC20PermitUpgradeable   is   Initializable,   ERC20Upgradeable,
IERC20PermitUpgradeable, EIP712Upgradeable {

File: WithBlockedList.sol
19: contract WithBlockedList is OwnableUpgradeable {
```

Therefore, the following is enough:

Unset

```
File: TetherToken.sol
23: contract TetherToken is
- 24:     Initializable,
25:     ERC20PermitUpgradeable,
- 26:     OwnableUpgradeable,
27:     WithBlockedList,
28:     EIP3009
29: {
```

G-2. Unchecking arithmetics operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

Consider wrapping with an unchecked block where it's certain that there cannot be an underflow

25 gas saved per instance

Affected code:

- contracts/Wrappers/FeeCurrencyWrapper.sol

```
Unset
# File: contracts/Wrappers/FeeCurrencyWrapper.sol

FeeCurrencyWrapper.sol:48:     digitDifference = decimals - wrappedTokenDigits;

FeeCurrencyWrapper.sol:114:         uint256 roundingError = debited - (refundAmountScaled +
tipAmountScaled + baseFeeAmountScaled);
```

feeCurrency wrapper has constant gas cost for debit and credit

G-3. Use calldata instead of memory for function arguments that do not get mutated

When a function with a memory array is called externally, the abi.decode() step has to use a for-loop to copy each index of the calldata to the memory index. Each iteration of this for-loop costs at least 60 gas (i.e. 60 * <mem_array>.length). Using calldata directly bypasses this loop.

If the array is passed to an internal function which passes the array to another internal function where the array is modified and therefore memory is used in the external call, it's still more gas-efficient to use calldata when the external function uses modifiers, since the modifiers may

prevent the internal functions from being called. Structs have the same overhead as an array of length one.

Saves 60 gas per instance

Affected code:

- contracts/Tether/TetherToken.sol

```
Unset
# File: contracts/Tether/TetherToken.sol

TetherToken.sol:198:        address[] memory _recipients,

TetherToken.sol:199:        uint256[] memory _values
```

changed

G-4. Use Custom Errors instead of Revert Strings to save Gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas

Additionally, custom errors can be used inside and outside of contracts (including interfaces and libraries).

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., revert("Insufficient funds.");), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Consider replacing all revert strings with custom errors in the solution, and particularly those that have multiple occurrences:

Affected code:

- contracts/Tether/TetherToken.sol

```
Unset
# File: contracts/Tether/TetherToken.sol

TetherToken.sol:91:              require(block.timestamp <= deadline, "ERC20Permit: expired
deadline");

TetherToken.sol:274:        require(!isBlocked[from], "Sender is not blocked");

TetherToken.sol:331:        require(!isBlocked[from], "Sender is not blocked");

TetherToken.sol:393:        require(!isBlocked[from], "Sender is not blocked");

TetherToken.sol:452:        require(!isBlocked[from], "Sender is not blocked");
```

- contracts/Wrappers/ArbitrumExtension.sol

```
Unset
# File: contracts/Wrappers/ArbitrumExtension.sol

ArbitrumExtension.sol:43:     require(msg.sender == l2Gateway, "ONLY_GATEWAY");

ArbitrumExtension.sol:54:     require(_l2Gateway != address(0), "INVALID_GATEWAY");

ArbitrumExtension.sol:55:     require(l2Gateway == address(0), "ALREADY_INIT");
```

- contracts/Wrappers/CeloExtension.sol

```
Unset
# File: contracts/Wrappers/CeloExtension.sol

CeloExtension.sol:29:              require(msg.sender   ==   address(0)   ||   msg.sender   ==
feeCurrencyWrapper, "Only VM can call");
```

- contracts/Wrappers/FeeCurrencyWrapper.sol

```
Unset
# File: contracts/Wrappers/FeeCurrencyWrapper.sol

FeeCurrencyWrapper.sol:13:     require(msg.sender == address(0), "Only VM can call");
```

```
FeeCurrencyWrapper.sol:75:     require(toDebit > 0, "Can not debit 0");
```

- contracts/Wrappers/WrappedExtension.sol

Unset

```
# File: contracts/Wrappers/WrappedExtension.sol

WrappedExtension.sol:68:       require(balance > 0, "No balance");

WrappedExtension.sol:76:       require(difference > 0, "No difference");
```

G-5. `public` functions not called by the contract should be declared `external` instead

Affected code:

- contracts/Tether/TetherToken.sol

Unset

```
# File: contracts/Tether/TetherToken.sol

TetherToken.sol:197:     function multiTransfer(

TetherToken.sol:210:     function mint(address _destination, uint256 _amount) public onlyOwner
{

TetherToken.sol:215:     function redeem(uint256 _amount) public onlyOwner {

TetherToken.sol:220:     function destroyBlockedFunds(address _blockedUser) public onlyOwner {

TetherToken.sol:263:     function transferWithAuthorization(

TetherToken.sol:382:     function receiveWithAuthorization(
```

- contracts/Wrappers/FeeCurrencyWrapper.sol

```
Unset
# File: contracts/Wrappers/FeeCurrencyWrapper.sol

FeeCurrencyWrapper.sol:56:   function balanceOf(address account) public view returns (uint256)
{

FeeCurrencyWrapper.sol:64:   function totalSupply() public view returns (uint256) {
```

- contracts/Wrappers/WrappedExtension.sol

```
Unset
# File: contracts/Wrappers/WrappedExtension.sol

WrappedExtension.sol:46:     function combinedTotalSupply() public view returns (uint256) {

WrappedExtension.sol:52:     function deposit(uint _value) public returns (bool success) {

WrappedExtension.sol:58:     function withdraw(uint _value) public returns (bool success) {

WrappedExtension.sol:65:      function withdrawDifferentToken(address _differentToken) public
onlyOwner returns (bool) {

WrappedExtension.sol:74:       function withdrawBalanceDifference() public onlyOwner returns
(bool success) {

WrappedExtension.sol:84:      function toggleWithdrawable(bool _canWithdraw) public onlyOwner
{
```

G-6. Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

Up until Solidity 0.8.13: `!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

Consider changing `> 0` with `!= 0` and enabling the Optimizer.

Affected code:

- contracts/Wrappers/FeeCurrencyWrapper.sol

```
Unset
# File: contracts/Wrappers/FeeCurrencyWrapper.sol

FeeCurrencyWrapper.sol:75:     require(toDebit > 0, "Can not debit 0");
```

- contracts/Wrappers/WrappedExtension.sol

```
Unset
# File: contracts/Wrappers/WrappedExtension.sol

WrappedExtension.sol:68:        require(balance > 0, "No balance");

WrappedExtension.sol:76:        require(difference > 0, "No difference");
```

G-7. Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Affected code:

- contracts/Tether/TetherToken.sol

```
Unset
# File: contracts/Tether/TetherToken.sol

TetherToken.sol:205:        for (uint256 i = 0; i < _recipients.length; i++) {
```

G-8. `++i` costs less gas compared to `i++`

Pre-increments are cheaper.

For a `uint256 i` variable, the following is true with the Optimizer enabled at 10k:

Increment:

- `i += 1` is the most expensive form
- `i++` costs 6 gas less than `i += 1`
- `++i` costs 5 gas less than `i++` (11 gas less than `i += 1`)

Saves 5 gas per instance

Affected code:

- contracts/Tether/TetherToken.sol

```
Unset
# File: contracts/Tether/TetherToken.sol

TetherToken.sol:205:        for (uint256 i = 0; i < _recipients.length; i++) {
```

G-9. Increments/decrements can be unchecked in for-loops

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

The change would be:

```
Unset
- for (uint256 i; i < numIterations; i++) {
+ for (uint256 i; i < numIterations;) {
  // ...
+   unchecked { ++i; }
}
```

These save around 25 gas saved per instance.

The same can be applied with decrements (which should use `break` when `i == 0`).

The risk of overflow is non-existent for uint256.

Affected code:

- contracts/Tether/TetherToken.sol

```
Unset
# File: contracts/Tether/TetherToken.sol

TetherToken.sol:205:        for (uint256 i = 0; i < _recipients.length; i++) {
```

certora

tether

Formal Verification

Assumptions and Simplifications

General Assumptions

-

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Violated	A counter-example exists that violates one of the assertions of the rule.

Formal Verification Properties

TetherToken.sol

Assumptions

- Owner can make all operations and transfers even if blocked.
- Blocked addresses can receive funds.
- Owner can make multiple actions which are risky - self block & destroy, self block & redeem
- Signature is unique only between every 2 users, but for 2 contracts or user & contract we can find violations

Properties

ID	Rule Name	Description
1	Verified **InitializeDoesntChangeAfterInit**	**Initialized flags never change after init**
2	Verified **cantCallInitilizeTwice**	**Can't initialize twice**
3	Verified **contractOwnerDosentChange**	**Only owner can change ownership**
4	Verified **ownerIsZeroAddressOnlyWhenUseRenounceOwnership**	**Owner is zero address only if ownership was renounced**
5	Verified **onlyOwnerCanChangeOwnership**	**Only owner can change ownership**
6	Verified **changeOwnershipDenialOfService**	**DOS check for ownership changes (Note: no two step ownership change -**

		L4)
7	Verified **OnlyOwnerModifierIntegrity**	**Integrity of onlyOwner modifier**
8	Verified **renouncedOwnershipCannotRecover**	**Cannot recover renounced ownership**
9	Verified **onlyOwnerMintOrRedeem**	**Only owner can mint or redeem**
10	Verified **totalSupplyIsSumOfBalances**	**Total supply is sum of balances**
11	Verified **totalSupplyNeverOverflow**	**totalSupply cannot overflow**
12	Verified **onlyAllowedMethodsMayChangeTotalSupply**	**Only specific methods can change totalSupply**
13	Verified **onlyMultiTransferCanChangeMoreThanTwoBalances**	**Only multiTransfer can change the balance of more than 2 addresses**
14	Verified **onlyAllowedMethodsMayChangeBalance**	**Only specific methods can change balance**
15	Verified **onlyAuthorizedCanTransfer**	**Only authorized addresses can transfer (Note: unrelated to blocked addresses)**

16	Verified **onlyAuthorizedCanTransferIntegrity**	**Transfer integrity**
17	Verified **authorizedTransferDoesNotAffectThirdParty**	**Third party unaffected by transfers**
18	Verified **transferWithAuthorizationRevertCondition**	**Transfer with authorization reverting conditions**
19	Verified **onlyAuthorizedCanReceiveIntegrity**	**Only authorized addresses can receive**
20	Verified **authorizedReceiveDoesNotAffectThirdParty**	**Receiving does not affect third party**
21	Verified **receiveWithAuthorizationRevertCondition**	**Receive with authorization reverting conditions**
22	Verified **transferIntegrity**	**Transfer integrity**
23	Verified **transferIsOneWayAdditive**	**Transfer is additive**
24	Verified **transferRevertingCondition**	**Transfer reverting conditions**
25	Verified **transferDoesNotAffectThirdParty**	**Third party unaffected by transfer**
26	Verified	**TransferFrom integrity**

	transferFromIntegrity	
27	Verified **transferFromIsOneWayAdditive**	**TransferFrom is additive**
28	Verified **transferFromRevertingCondition**	**TransferFrom reverting conditions**
29	Verified **transferFromDoesNotAffectThirdParty**	**Third party unaffected by transferFrom**
30	Verified **onlyAllowedMethodsMayChangeAllowance**	**Only specific methods can change allowance**
31	Verified **onlyHolderOrSpenderCanChangeAllowance**	**Only holder or spender can change allowance**
32	Verified **approveIntegrity**	**Integrity of approve**
33	Verified **approveRevertingConditions**	**Approve reverting conditions**
34	Verified **approveDoesNotAffectThirdParty**	**Third party unaffected by approve**
35	Verified **mintIntegrity**	**Integrity of mint**
36	Verified **mintRevertingConditions**	**Mint reverting conditions**

37	Verified **mintDoesNotAffectThirdParty**	**Third party unaffected by mint**
38	Verified **redeemIntegrity**	**Integrity of redeem**
39	Verified **redeemRevertingConditions**	**Redeem reverting conditions**
40	Verified **redeemDoesNotAffectThirdParty**	**Third party unaffected by redeem**
41	Verified **permitIntegrity**	**Integrity of permit**
42	Verified **permitDoesNotAffectThirdParty**	**Third party unaffected by permit**
43	Verified **permitRevertConditions**	**permit reverting conditions**
44	Verified **signatureIsUniquePerHolder**	**Uniqueness of signature (Note: Violated in the case where holder is user address and another holder is a contract address)**
45	Verified **userCanBeAddedToBlocklistOnlyUsingAddToBlockList**	**Only addToBlockList() can add to block list**
46	Verified **onlyOwnerCanAddUserToBlockList**	**Only owner can block**
47	Verified **onlyOwnerCanRemoveUserFromBlo**	**Only owner can unblock**

	ckList	
48	Verified **userRemovedFromBlocklistOnlyUsingRemoveFromBlockedList**	**Only removeFromBlockList() can unblock**
49	Verified **blockedUserBalanceCannotDecrease**	**Blocked balances cannot decrease (Note: unless owner and calls the functions)**
50	Verified **cantDestroyUnblockedUserFunds**	**Unblocked user's funds are safe from destruction**
51	Verified **contractAllowanceAlwaysZero**	**Contract's allowance is always zero**
52	Verified **reentrancySafety**	**Re-entrancy safety meaning that all storage accesses occur either before external call or after**
53	Verified **msgSenderIsNotCurrentContract**	**CurrentContract cannot be "message sender".**
54	Verified **checkHolderIsNotCurrentContract**	**Using currentContract as the holder, cannot perform - permit, transferWithAuthorization, receiveWithAuthorization.**
55	Verified **NoDiffBetweenPermitImplementations NoDiffBetweenReceiveImplementations NoDiffBetweenTransferImplementations**	**Using signature or v,r,s methods are giving the same outcome.**

56	Verified **contractBalanceAlwaysZero**	**Celo contract balance remains zero as an invariant after every operation.**

WrappedExtension.sol

Assumptions

- Owner not call the following function withdrawDifferentToken(address _differentToken) public onlyOwner returns (bool) with the currentContract Address

Properties

ID	Rule Name	Description
1	Verified **initializeDoesntChangeAfterInit**	**Initialized flags never change after init**
2	Verified **cantCallInitilizeTwice**	**Can't initialize twice**
3	Verified **ownerIsZeroAddressOnlyWhenUseRenounceOwnership**	**Owner is zero address only if ownership was renounced**
4	Verified **totalSupplyNeverOverflow**	**totalSupply cannot overflow**
5	Verified **onlyMultiTransferCanChangeMoreThanTwoBalances**	**Only multiTransfer can change the balance of more than 2 addresses**
6	Verified **onlyAuthorizedCanTransfer**	**Only authorized addresses can transfer (Note: unrelated to blocked addresses)**

7	Verified **onlyAuthorizedCanTransferIntegrity**	**Transfer integrity**
8	Verified **authorizedTransferDoesNotAffectThirdParty**	**Third party unaffected by transfers**
9	Verified **onlyAuthorizedCanReceiveIntegrity**	**Only authorized addresses can receive**
10	Verified **authorizedReceiveDoesNotAffectThirdParty**	**Receiving does not affect third party**
11	Verified **transferIntegrity**	**Transfer integrity**
12	Verified **transferIsOneWayAdditive**	**Transfer is additive**
13	Verified **transferRevertingCondition**	**Transfer reverting conditions**
14	Verified **transferDoesNotAffectThirdParty**	**Third party unaffected by transfer**
15	Verified **transferFromIntegrity**	**TransferFrom integrity**
16	Verified **transferFromIsOneWayAdditive**	**TransferFrom is additive**
17	Verified **transferFromRevertingCondition**	**TransferFrom reverting conditions**

18	Verified **transferFromDoesNotAffectThirdParty**	**Third party unaffected by transferFrom**
19	Verified **onlyAllowedMethodsMayChangeAllowance**	**Only specific methods can change allowance**
20	Verified **approveIntegrity**	**Integrity of approve**
21	Verified **approveRevertingConditions**	**Approve reverting conditions**
22	Verified **approveDoesNotAffectThirdParty**	**Third party unaffected by approve**
23	Verified **mintIntegrity**	**Integrity of mint**
24	Verified **mintRevertingConditions**	**Mint reverting conditions**
25	Verified **mintDoesNotAffectThirdParty**	**Third party unaffected by mint**
26	Verified **redeemIntegrity**	**Integrity of redeem**
27	Verified **redeemRevertingConditions**	**Redeem reverting conditions**
28	Verified **redeemDoesNotAffectThirdParty**	**Third party unaffected by redeem**
29	Verified	**Integrity of permit**

	permitIntegrity	
30	Verified **permitDoesNotAffectThirdParty**	**Third party unaffected by permit**
31	Verified **userCanBeAddedToBlocklistOnlyUsingAddToBlockList**	**Only addToBlockList() can add to block list**
32	Verified **onlyOwnerCanAddUserToBlockList**	**Only owner can block**
33	Verified **onlyOwnerCanRemoveUserFromBlockList**	**Only owner can unblock**
34	Verified **userRemovedFromBlocklistOnlyUsingRemoveFromBlockedList**	**Only removeFromBlockList() can unblock**
35	Verified **cantDestroyUnblockedUserFunds**	**Unblocked user's funds are safe from destruction**
36	Verified **blockedUserWrappedBalanceCannotDecrease**	**Blocked balances on wrapped contract cannot decrease**
37	Verified **msgSenderIsNotCurrentContract**	**CurrentContract cannot be "message sender".**
38	Verified **totalSupplyIsSumOfBalances**	**Total supply is sum of balances**
39	Verified **totalSuppliesRelations**	**Difference between total supplies only changes by specific functions…**

40	Verified **depositIntegrity**	**Deposit work and update storage as expected**
41	Verified **depositNotEffectedThirdParty**	**3rd party user won't affect from other users deposits**
42	Violated **blockedUserCanDepositBlockedFunds**	Violation reason: **If by mistake originalToken Owner is the wrapped contract, a user blocked in the originalToken will be able to send its blocked funds to the wrapped and get a "clean" new wrapped tokens. (INFO-6)**
43	Verified **depositRevertConditions**	**Specify all the causes for deposit reverts**
44	Verified **contractAllowanceAlwaysZero**	**Contract has zero allowances**
45	Verified **withdrawIntegrity**	**Withdraw works and update storage as expected**
46	Verified **blockedUserCantWithdraw**	**Block users can't call the withdraw function.**
47	Verified **withdrawNotEffectThirdPart**	**3rd party user won't affect from other users withdraws**
48	Verified **withdrawRevertCondition**	**Specify all the causes for withdraw reverts**
49	Verified **onlyWithdrawFunctionsCanEmptyWrappedBalance**	**All wrapped balance on original token can be withdrawn and only by using specific withdraw functions**

50	Verified **originalTokencontractAllowanceAlwaysZero**	**Original token has zero allowances**
51	Verified **`wrappedAllowanceAlwaysZeroInOriginalToken`**	**Wrapped contract has zero allowances in original token**
52	Verified **`roundTripDeposit`**	**User holdings don't increase on round trip deposit -> withdraw -> deposit**
53	Verified **`roundTripWithdraw`**	**User holdings don't increase on round trip withdraw -> deposit -> withdraw**
54	Violated **depositDenialOfService**	Violation reason: **If by mistake address zero is blocked, the deposit function will be out of service. (INFO-7)**
55	Verified **contractBalanceAlwaysZero**	**Wrapped token contract balance in wrapped is always zero**
56	Verified **blockedUserOriginalTokenBalanceCannotDecrease**	**Blocked balances on original Token cant decrease**
57	Verified **originalTokenBalanceAlwaysZero**	**Original token contract balance is always zero in Original token balance**

CeloExtension.sol

Assumptions

- A blocked user in Celo extension can still pay gas and the gas address itself can also transfer tokens **while blocked**.
- Debit and credit gas fees can decrease/increase balances.
- GAS_FEE_ADDRESS = 0x000...00Ce106A5.

Properties

ID	Rule Name	Description
1	Verified **initializeDoesntChangeAfterInit**	**Initialized flags never change after init**
2	Verified **cantCallInitilizeTwice**	**Can't initialize twice**
3	Verified **ownerIsZeroAddressOnlyWhenUseRenounceOwnership**	**Owner is zero address only if ownership was renounced**
4	Verified **totalSupplyNeverOverflow**	**totalSupply cannot overflow**
5	Verified **onlyMultiTransferCanChangeMoreThanTwoBalances**	**Only multiTransfer and creditGasFees can change the balance of more than 2 addresses**
6	Verified	**Only authorized addresses can transfer**

	onlyAuthorizedCanTransfer	**(Note: unrelated to blocked addresses)**
7	Verified **onlyAuthorizedCanTransferIntegrity**	**Transfer integrity**
8	Verified **authorizedTransferDoesNotAffectThirdParty**	**Third party unaffected by transfers**
9	Verified **onlyAuthorizedCanReceiveIntegrity**	**Only authorized addresses can receive**
10	Verified **authorizedReceiveDoesNotAffectThirdParty**	**Receiving does not affect third party**
11	Verified **transferIntegrity**	**Transfer integrity**
12	Verified **transferIsOneWayAdditive**	**Transfer is additive**
13	Verified **transferRevertingCondition**	**Transfer reverting conditions**
14	Verified **transferDoesNotAffectThirdParty**	**Third party unaffected by transfer**
15	Verified **transferFromIntegrity**	**TransferFrom integrity**
16	Verified **transferFromIsOneWayAdditive**	**TransferFrom is additive**
17	Verified	**TransferFrom reverting conditions**

	transferFromRevertingCondition	
18	Verified **transferFromDoesNotAffectThirdParty**	**Third party unaffected by transferFrom**
19	Verified **onlyAllowedMethodsMayChangeAllowance**	**Only specific methods can change allowance**
20	Verified **approveIntegrity**	**Integrity of approve**
21	Verified **approveRevertingConditions**	**Approve reverting conditions**
22	Verified **approveDoesNotAffectThirdParty**	**Third party unaffected by approve**
23	Verified **mintIntegrity**	**Integrity of mint**
24	Verified **mintRevertingConditions**	**Mint reverting conditions**
25	Verified **mintDoesNotAffectThirdParty**	**Third party unaffected by mint**
26	Verified **redeemIntegrity**	**Integrity of redeem**
27	Verified **redeemRevertingConditions**	**Redeem reverting conditions**
28	Verified	**Third party unaffected by redeem**

	redeemDoesNotAffectThirdParty	
29	Verified **permitIntegrity**	**Integrity of permit**
30	Verified **permitDoesNotAffectThirdParty**	**Third party unaffected by permit**
31	Verified **userCanBeAddedToBlocklistOnlyUsingAddToBlockList**	**Only addToBlockList() can add to block list**
32	Verified **onlyOwnerCanAddUserToBlockList**	**Only owner can block**
33	Verified **onlyOwnerCanRemoveUserFromBlockList**	**Only owner can unblock**
34	Verified **userRemovedFromBlocklistOnlyUsingRemoveFromBlockedList**	**Only removeFromBlockList() can unblock**
35	Verified **cantDestroyUnblockedUserFunds**	**Unblocked user's funds are safe from destruction**
36	Verified **blockedUserBalanceCannotDecrease_Celo**	**Blocked balances cannot decrease (Note: unless destroyBlockedFunds() called, or blocked is owner and calls redeem)**
38	Verified **totalSupplyIsSumOfBalances**	**Total supply is sum of balances**
39	Verified	**Only owner can change ownership**

	contractOwnerDosentChange	
40	Verified **onlyOwnerCanChangeOwnership**	**Only owner can change ownership**
41	Verified **changeOwnershipDenialOfService**	**DOS check for ownership changes (Note: no two step ownership change - L4)**
42	Verified **OnlyOwnerModifierIntegrity**	**Integrity of onlyOwner modifier**
43	Verified **onlyOwnerCanChangeOwnership**	**Only owner can change ownership**
44	Verified **debitGasFeesIntegrity**	**Integrity of debitGasFees**
45	Verified **creditGasFeesIntegrity**	**Integrity of creditGasFees (Note: GAS_FEE_ADDRESS is not credited and is different from all other addresses)**
46	Verified **canCreditDebited**	**Debit can be withdrawn from FeeCurrencyWrapper is exist**
47	Verified **creditGasFeesDenialOfService**	**If debitGasFees and some function after it didn't revert, creditGasFees will not always revert, denying service.**
48	Verified **celoVMFunctionExecution**	**Tries to model the celo VM behavior. Assuming GAS_FEE_ADDRESS balance is 0 and gas fees recipient and community funds are not the zero address, checks that after executing a function with**

		debitGasFees before and creditGasFees after, the balance of GAS_FEE_ADDRESS and FeeCurrencyWrapper.debited stays 0.
49	Violated **reentrancySafety**	**Re-entrancy safety meaning that all storage accesses occur either before external call or after. (INFO-8)** Violation reason: Violated for FeeCurrencyWrapper.creditGasFees. The external call is for wrappedToken.creditGasFees. It should be fine as long as wrappedToken=CeloExtension as there is trust between FeeCurrencyWrapper and CeloExtension.
50	Verified **msgSenderIsNotCurrentContract**	**CurrentContract cannot be "message sender".**
51	Verified **contractBalanceAlwaysZero**	**Celo contract balance remains zero as an invariant after every operation.**

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.