Security Assessment & Formal Verification Report

March 2024

Prepared for
Safe Global

Table of content

Project Summary

Project Scope

Repo Name	Repository	Commits	Compiler version	Platform
safe-locking	safe-locking	f467abf3d3f16796d71e80c323d2112aa002d82b	Solc 08.23	EVM

Project Overview

This document describes the specification and verification of the **Safe Global - safe-locking** using the Certora Prover and manual code review findings. The work was undertaken from {**4 March 2024** to **11 March 2024}**.

The following contract list is included in our scope:

```
contracts/base/TokenRescuer.sol
contracts/interfaces/ISafeTokenLock.sol
contracts/SafeTokenLock.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.

Protocol Overview

Safe locking contract facilitates locking Safe tokens. Users can lock and unlock tokens anytime and also withdraw after the `COOLDOWN_PERIOD` is over. The contract also provides feature controlled by the admin address to recover ERC20 tokens other than Safe tokens.

Findings Summary

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

Severity	Discovered	Acknowledged	Code Fixed
Critical	0		
High	0		
Medium	0		
Low	0		
Informational	0		
Total			

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Probability	

certora

Formal Verification

Assumptions and Simplifications

General Assumptions

A. We assume that all arrays are at most max(uint128) long.
B. The maximum possible timestamp corresponds to the beginning of the year 2525.

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

Safe Token Lock

Assumptions

- Any loop can iterate at most 3 times.

Properties

Rule Name	Description
safeTokenSelfBalanceIsZero	Verify that Safe token contract's Safe token balance is always zero.
safeTokenCannotLock	Verify that Safe token contract cannot lock tokens
contractCannotOperateOnItself	Invariant that proves that the Safe token locking contract never has a locked balance. i.e. there is no way for an external caller to get the locking contract to call \`lock\`, \`unlock\` or \`withdraw\` on itself
noAllowanceForSafeTokenLock	Invariant that proves that the Safe token locking contract never grants allowance to another address. i.e. there is no way for an external caller to get the locking contract to call \`approve\` or \`increaseAllowance\` on the Safe
contractBalanceIsGreaterThanTotalLockedAndUnlockedAmounts	Invariant proves that the locking contract's Safe token balance is always greater than the sum of all user's Safe token balance in the Safe locking contract
totalLockedIsGreaterThanUserLocked	Verify that total locked safe Is greater or equal to each user locked safe

totalUnlockedIsGreaterThanUserUnlocked	Verify that total unlocked safe Is greater or equal to each user unlocked safe
userTokenBalanceIsLessThanTotalSupply	Invariant that a user's Safe token balance in the locking contract is less that the total supply of Safe token.
unlockStartBeforeEnd	Invariant that the `unlockStart` index is always before the `unlockEnd` index for a user.
userUnlockedIsSumOfUnlockAmounts	Invariant that the unlocked amount for a user is always equal to the sum of the individual amounts of each of their pending unlocks.
unlockAmountsAreNonZero	Invariant to prove that no unlock amount in the list is 0
addressZeroCannotLock	Invariant that proves that the user token balance of the zero address in the locking contract is always zero.
unlocksAreOrderedByMaturityTimestamp	Invariant to prove that unlock maturity timestamp is always increasing.
configurationNeverChanges	Verify that the `SAFE_TOKEN` and `COOLDOWN_PERIOD` are immutable
getUserNeverReverts	Verify that the `getUser` function never reverts
getUserTokenBalanceNeverReverts	Verify that the `getUserTokenBalance` function never reverts

getUnlockNeverReverts	Verify that the `getUnlock` function never reverts
doesNotAffectOtherUserBalance	Verify that no operations on the Safe token locking contract done by user A can affect the Safe token balance of user B in the locking contract
ownerCanAlwaysTransferOwnership	Verify that an owner can always transfer ownership
pendingOwnerCanAlwaysAcceptOwnership	Verify that a pending owner can always accept ownership after transferOwnership
ownerCanAlwaysRenounceOwnership	Verify that an owner can always renounce ownership
onlyOwnerOrPendingOwnerCanChangeOwner	Verify that only the `owner` (when renouncing ownership) and the `pendingOwner` (when accepting ownership) can change the value of the contract `owner`
onlyOwnerOrPendingOwnerCanChangePendingOwner	Verify that only the `owner` (when transferring or renouncing ownership) and the `pendingOwner` (when accepting ownership) can change the value of the contract `pendingOwner`
canAlwaysLock	Verify that the user can always lock tokens. Notable exceptions are not having enough allowance to locking contract, not having enough balance, passed amount being zero and the Safe token contract being paused
allLockedCanGetUnlocked	Verify that the user can always unlock

	tokens. If locked tokens are less than before, then unlocked tokens are more by exactly the difference than before
canAlwaysUnlock	Verify that a user can always unlock their tokens. Notable exceptions are documented below
unlockMaturityTimestampDoesNotChange	Verify that it is impossible for a user to modify the time at which their unlock matures and can be withdrawn
cannotUnlockPastMaxUint32	Verify that it is impossible to unlock more tokens once `unlockEnd` has reached the maximum value that can be represented by a `uint32`
unlockIndexShouldReturnLastEndIndex	Verify that index received from `unlock` is always the last `unlockEnd`
cannotWithdrawMoreThanUnlocked	Verify that withdrawal cannot increase the balance of a user more than their total unlocked amount, i.e. it is impossible to withdraw tokens without having previously unlocked them
withdrawAmountCorrectness	Verify that withdrawing returns the exact amount of tokens that were transferred out and the user total amounts are correctly updated
cannotWithdrawBeforeCooldown	Verify that unlock tokens can only be withdrawn once they mature
withdrawIsCommutative	Verify that withdrawing is commutative. That is, withdrawing with `maxUnlocks` of `n`

	then `m`, is equivalent to `m` then `n`
alwaysPossibleToWithdraw	Verify that it is always possible to, given an initial state with some locked token amount, to fully withdraw the entire locked balance
withdrawShouldAlwaysIncreaseReceiverTokenBalance	Verity that the receiver's token balance always increases after a successful withdrawal
withdrawReturnsValueBasedOnMaturedUnlock	Verify that the `withdraw` function returns the correct amount based on the user's matured unlocks
canAlwaysWithdrawEverythingAfterMaturity	Verify that the locked amount can always be withdrawn after maturity
noFrontRunning	Verify that it is not possible to front-run, except certain scenarios
lockIsCommutative	Verify that lock Is Commutative
unlockIsCommutative	Verify that unlock Is Commutative (locked and unlocked amounts wise)
approveDoesNotAffectThirdParty	Verify that approve doesn't affect third party

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.