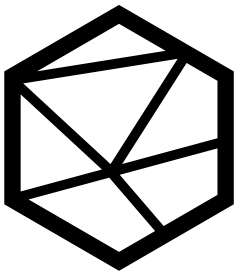
Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



57BLOCKS

Stellar Timelock Contract

Veridise Inc.

August 6, 2024

► **Prepared For:**

57Blocks
https://57blocks.io

► **Prepared By:**

Evgeniy Shishkin
Alberto Gonzalez

► **Contact Us:**

contact@veridise.com

► **Version History:**

Aug. 6, 2024	V2
Jul. 18, 2024	V1
Jul. 16, 2024	Initial Draft

Contents

1 Executive Summary

From July 9, 2024 to July 11, 2024, 57Blocks engaged Veridise to conduct the program code audit of their Stellar Timelock Contract project. The audit covered the logic behind a Timelock contract, including a role-based access control system.

Veridise conducted the audit over 6 person-days, with 2 security analysts reviewing code over 3 days on commit f143245. The auditing strategy involved extensive manual code review performed by Veridise security analysts.

Project overview. In various decentralized applications (DApps), there is a common issue where administrative actions can be taken at any time by privileged users, leaving regular users unable to respond quickly if they object to the action. This issue can undermine trust in DApps and introduce additional security risks. To make things more manageable, the Timelock mechanism has been introduced.

The Timelock is a middleman between a party initiating a smart contract operation and a destination smart contract itself. Its purpose is to introduce a reasonable delay between the initiation of the operation and its execution in the smart contract. This delay allows DApp users or token holders to critically evaluate the implications of the issued operation before it gets executed. Stellar Timelock Contract is concrete implementation of such Timelock mechanism, heavily inspired by OpenZeppelin's implementation.

The workflow for applying Stellar Timelock Contract to a token contract is described below.

A token owner deploys the Stellar Timelock Contract instance specifying the following parameters:

- Owner - the *Owner* is a privileged user who is able to assign and revoke the roles of proposer, canceller, and executor to other users.
- Minimum delay - the minimum time interval that should be observed after submitting and before executing a privileged operation. During this time, regular users have a chance to critically assess the implications of the submitted operation.
- Proposers - a set of user addresses that are assigned *Proposer* and *Canceller* roles right at the initialization step.
- Executors - a set of user addresses that are assigned *Executor* roles right at the initialization step.

The token contract owner changes the token contract ownership from their own address to the address of the deployed Stellar Timelock Contract, effectively relinquishing their ownership role.

From that point, users with specific roles can *schedule*, *execute* or *cancel* privileged requests to the destination token contract by issuing corresponding *operations* into the Stellar Timelock Contract.

For example, to execute the token mint operation, the operation has to be submitted to Stellar Timelock Contract by a user with the *Proposer* role, specifying a time interval that should be observed before the operation can be executed.

After the specified time has passed, any user who has the *Executor* role can ask Stellar Timelock Contract to execute the submitted operation using all the same parameters that were used to submit it.

If the submitted operation becomes irrelevant, any user with the *Canceller* role can cancel it.

Please note the following nuances:

- The *Owner* of the Timelock is a privileged entity that is able to grant and revoke *Proposer*, *Canceller* and *Executor* roles, as well as changing the *minimum delay* value. If the *Owner*'s keys are compromised, it could have serious consequences for the contract being managed.
- The submitted operation is not guaranteed to be executed on time, or executed at all.

Code assessment. The developers provided the source code of the Stellar Timelock Contract contracts for the code audit. The source code appears to be mostly original code written by the Stellar Timelock Contract developers. However, the source code is based on the OpenZeppelin versions of the TimelockController* and AccessControl† contracts. To facilitate the Veridise security analysts' understanding of the code, the Stellar Timelock Contract developers shared documentation in the form of READMEs and documentation comments on functions and storage variables.

The source code included a test suite that effectively covered aspects of the Stellar Timelock contract related to both role-based access control and the main Timelock operations.

Summary of issues detected. The code audit uncovered 4 issues. The most important were 2 low-severity issues: a centralization risk (V-STL-VUL-001) and insufficient operation delay validation (V-STL-VUL-002). The security analysts also identified 1 warning and 1 informational finding. The reported issues are not yet acknowledged by the developers.

Recommendations. After conducting the code audit for the protocol, the security analysts had a few suggestions to improve the Stellar Timelock Contract.

It is recommended to improve the documentation in the following way:

- Outline the differences between the Stellar Timelock Contract and the OpenZeppelin implementation. For instance, comparing to OpenZeppelin Timelock, the Stellar Timelock Contract does not contain batch operations processing.
- It is recommended to add advise on how to select the right number of privileged users for different scenarios.

* https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/governance/TimelockController.sol
† https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise 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.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Stellar Timelock Contract	f143245	Rust	Soroban

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
July 9 - July 11, 2024	Manual	2	6 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	2	2	1
Warning-Severity Issues	1	1	0
Informational-Severity Issues	1	1	1
TOTAL	4	4	2

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Access Control	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a code audit of Stellar Timelock Contract's smart contracts. The security analysis aimed to answer questions such as:

- Are there any common Soroban implementation flaws, such as incorrect storage management, incorrect build order, or others?
- Can the logic deviations from the OpenZeppelin implementation cause issues?
- Can the executor, proposer, and canceller perform actions beyond their expected behavior?
- Is there a way for a malicious user to prevent the scheduling, execution, or cancellation of operations outside of the access control system?
- Is it possible that a mistake in operations scheduling could cause a denial of service for the system?
- Is it possible to execute an operation before its deadline?

3.2 Audit Methodology & Scope

Scope. The scope of this code audit was limited to the following contracts:

- `time_lock/src/contract.rs`
- `time_lock/src/lib.rs`
- `time_lock/src/role_base.rs`
- `time_lock/src/time_lock.rs`

Methodology. Veridise security analysts inspected the provided tests, and read the Stellar Timelock Contract documentation. They then began a manual review of the code taking into consideration the OpenZeppelin's contracts.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our security analysts weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-STL-VUL-001	Centralization Risk	Low	Fixed
V-STL-VUL-002	Operation delay bounds are not enforced	Low	Partially Fixed
V-STL-VUL-003	Number of proposers is not enforced	Warning	Acknowledged
V-STL-VUL-004	Function name not validated in administrative o. . .	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-STL-VUL-001: Centralization Risk

Severity	Low	**Commit**	f143245
Type	Access Control	**Status**	Fixed
File(s)	`contract.rs`		
Location(s)	See the description		
Confirmed Fix At	4e24b47		

The timelock contract appears to closely follow the OpenZeppelin timelock contract implementation as its reference. According to the original OpenZeppelin design, during initialization, the deployer provides a user address to be assigned an admin role. This admin role is intended to be renounced at some point to conduct all admin operations through timelocked proposals, thereby avoiding the concentration of power in a single user's hands - a key aspect of the OpenZeppelin design.

However, the current timelock contract lacks this feature. While the owner could technically relinquish control by transferring ownership to a dummy address such as address(0), this would render all administrative functions, including `grant_role()`, `revoke_role()`, `update_min_delay()`, and `update_owner()` inaccessible.

Impact The current implementation does not allow for the removal of the owner, thereby introducing a risk of centralization.

Recommendation It is recommended to enable the execution of administrative functions by scheduling them as operations.

Developer Response Developers implemented the recommended fix.

4.1.2 V-STL-VUL-002: Operation delay bounds are not enforced

Severity	Low	**Commit**	f143245
Type	Data Validation	**Status**	Partially Fixed
File(s)	`contract.rs`		
Location(s)	initialize(), update_min_delay() , schedule()		
Confirmed Fix At	0e0310b		

The timelock contract employs a globally defined value called the minimum delay that the delay for submitting operations cannot be less than. The initialization logic enforces this value to be greater than `0` since a delay of `0` means the operation can be executed in the same block in which it was published, contradicting the very purpose of the timelock contract.

The timelock contract allows the minimum delay value to be changed later by calling the `update_min_delay()` function. However, this function does not include this check.

Impact The following problems were identified for this part of the code:

- Setting the minimum delay that is only slightly greater than `0` does not provide a reasonable timeframe for users to analyze the submitted operation.
- The minimum delay value is not verified within the `update_min_delay()` function.
- The delay value is not compared against a maximum value, allowing the submission of operations with unreasonably long time-frames.

Recommendation It is recommended to implement checks for both the minimum delay for the `update_min_delay()` function, as well as introduce the maximum delay value.

Developer Response We reviewed Openzeppelin's code and found that they didn't implement such minimum and maximum check for delay. However, considering that a delay that is too long could result in the entire system being unmodifiable for extended period, we have added a limit to the maximum value.

4.1.3 V-STL-VUL-003: Number of proposers is not enforced

Severity	Warning	**Commit**	f143245
Type	Data Validation	**Status**	Acknowledged
File(s)	`contract.rs`		
Location(s)	grant_role() , revoke_role()		
Confirmed Fix At			

During the initialization, the timelock contract grants special `proposer` and `canceller` permissions, as well as `executor` permissions to a group of distinguished users. From a security perspective, it is a good idea to limit the total number of such users to a reasonable amount. This limitation is enforced by a check in the `initialize()` function, which will not allow more than `MAX_ACCOUNTS_NUM` user accounts to be passed during initialization.

However, a similar check is not employed in the `grant_role()` function. Hence, while the initial set of privileged users may be small, the subsequent `grant_role()` calls can significantly increase and make it less manageable over time.

Additionally, if the aim is to make the timelock contract ownerless, as was the original design intention of the OpenZeppelin reference contract, it is important to also check for the lower bound of total granted roles. Having too few privileged users can also harm the system.

Impact Allowing too many or too few users propose, cancel or execute operations in the time-lock contract may affect its stability and security.

Recommendation It is recommended to implement upper and lower bound checks for the total number of proposer, canceler, and executor roles in the `grant_role()` and `revoke_role()` functions.

Developer Response Openzeppelin still has not added checks for the number of roles(prospsers, executors). We believe that the managment of roles should be handled by the users of the timelock cotract themselves.

4.1.4 V-STL-VUL-004: Function name not validated in administrative operations

Severity	Info	**Commit**	f143245
Type	Data Validation	**Status**	Fixed
File(s)	`contract.rs`		
Location(s)	schedule()		
Confirmed Fix At	72d7a82		

The timelock contract permits the scheduling of administrative operations, such as:

- `update_min_delay`
- `grant_role`
- `revoke_role`
- `update_owner`

To determine whether an operation is administrative, the contract checks if the operation's destination address matches the timelock contract's address, and if so, it enqueues the operation accordingly. However, the `fn_name` field, which specifies the expected function call, is not verified in any way.

Impact Submitting an administrative operation with a misspelled function name can lead to the operation being reverted, resulting in a loss of time.

Recommendation Since the timelock contract contains only four functions, it is advisable to explicitly check that the `fn_name` parameter within the `schedule` function matches one of these four functions.

Developer Response Developers implemented the recommended fix.

