Zellic

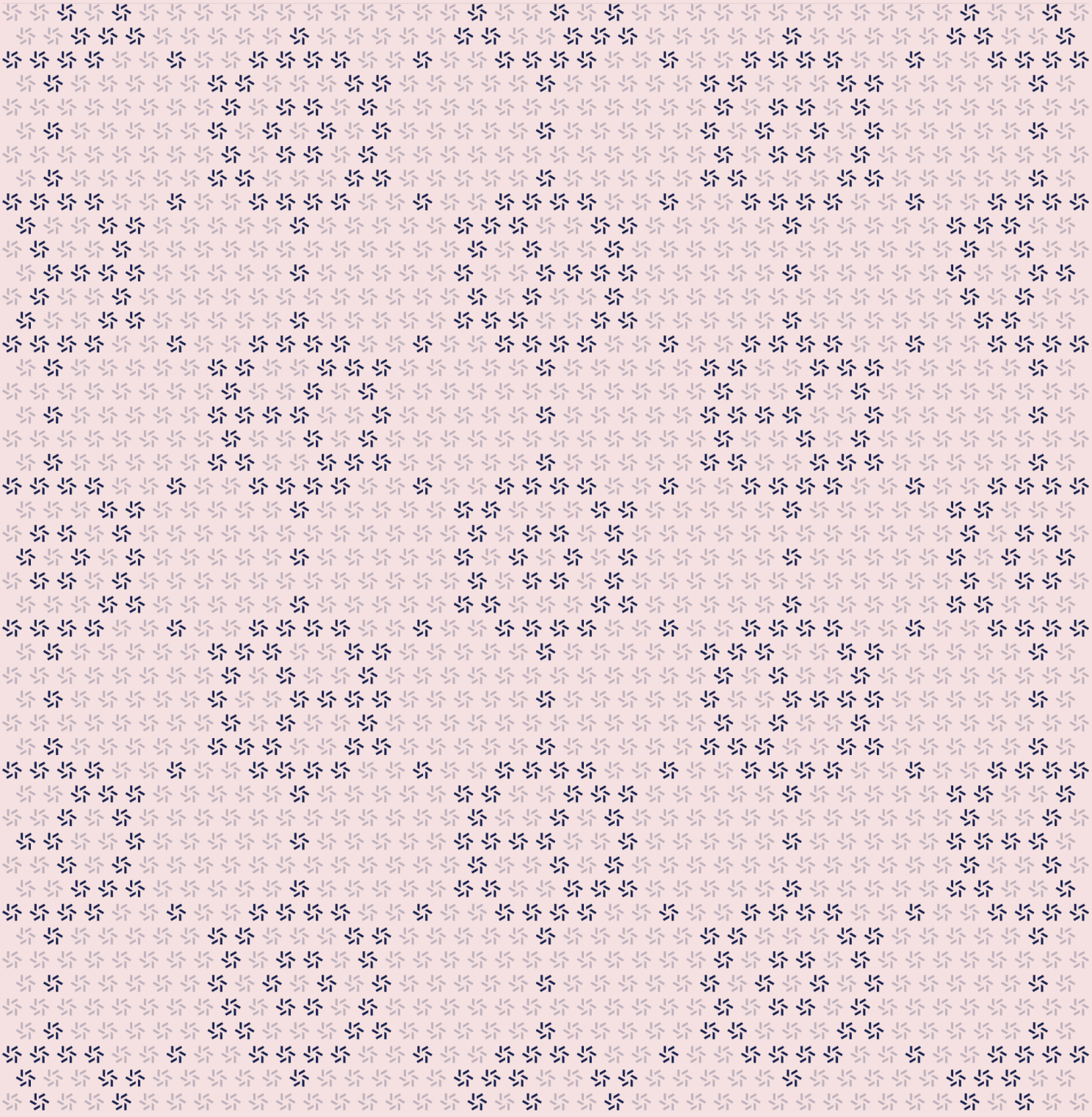
Prepared for
George Melika
Volta Circuit

Prepared by
Qingying Jie
Ayaz Mammadov
Zellic

January 30, 2026

Wallet

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Volta Circuit from January 23rd to January 27th, 2026. During this engagement, Zellic reviewed Wallet's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any ways in which owners could lose funds or control?
- Does the implementation of the contract meet expectations?
- Is the access control of important functions appropriate?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Wallet files, we discovered one finding, which was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Volta Circuit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	1

2. Introduction

2.1. About Wallet

Volta Circuit contributed the following description of Wallet:

> Volta is an M-of-N multisignature smart contract for the Stellar Soroban platform. It enables a group of owners to collectively manage on-chain actions and assets through a proposal-and-vote mechanism, where a configurable threshold of approvals is required before any action can be executed.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the files.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect

its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped files itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Wallet Files

Type	Rust
Platform	Soroban
Target	smart-contract-soroban
Repository	https://github.com/VoltaHQ/smart-contract-soroban ↗
Version	`c93d8c9493394f62efbd531ec328b953c6e2f2c8`

Programs	`events.rs`
	`lib.rs`
	`types.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of four person-days. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

The following consultants were engaged to conduct the assessment:

Qingying Jie
Engineer
qingying@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 23, 2026	Kick-off call
January 23, 2026	Start of primary review period
January 27, 2026	End of primary review period

3. Detailed Findings

3.1. The function `exists` cannot determine if the address is callable

Target	types.rs		
Category	Business Logic	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The function `validate`, when validating the data of an `Invoke` proposal, uses the function `exists` to check whether the calling address exists on the ledger.

```
impl Validate for Call {
    fn validate(&self, _env: &Env, _existing_config: Option<&Config>) ->
    Result<(), ContractError> {
        if !self.address.exists() {
            return Err(ContractError::AddressNotOnLedger);
        }

        // [...]
    }
}
```

Impact

The function `exists` cannot determine if the address is callable, and the call may fail if the proposal is approved.

Recommendations

In addition to checking whether the address exists, the function `executable` can also return the account type of the address. Consider using the function `executable` for the address verification.

Remediation

This issue has been acknowledged by Volta Circuit, and a fix was implemented in commit `6bc166b4` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. An unnecessary storage update

When a proposal is approved, it will be removed from the storage. So there is no need to update its data in storage before removing it. We recommend removing the unnecessary update to save on gas.

```
pub fn vote(
    env: Env,
    owner: Address,
    proposal_id: u64,
    vote: VoteType,
) -> Result<Proposal, ContractError> {
    // [...]

    let event_type: Option<Symbol> = match proposal.status {
        // [...]
        ProposalStatus::Approved => {
            proposal.set(&env);
            // [...]
            proposal.remove(&env);

            Some(symbol_short!("exec_prop"))
        }
```

This issue has been acknowledged by Volta Circuit, and a fix was implemented in commit 0a65354a ↗.

4.2. Abstention equivalence to "no"

Due to the fixed threshold system in the m-of-n wallet, an abstaining vote is equivalent to a no vote. In other governance/voting systems, usually an abstention reduces the number of votes to cross the threshold as the threshold is a percentage of nonabstaining votes. The reason abstention is included in this wallet is to allow owners to signal their intentions towards a proposal, specifically differentiating abstention from a no vote. It is recommended to inform users of this in the documentation.

4.3. Support only for auth entries from the volta wallet

The wallet is designed in such a way that all auth entries supplied will be signed by the volta wallet, so if there are any subcalls that require an auth entry signed by another user, they cannot be supplied as an extra parameter. As such, there are certain proposals that are not possible as they require auth entries from other users.

4.4. Upgrade contract validation

As of the time of writing, there is no way in Soroban Stellar SDK to verify that a WASM hash has been uploaded to the smart contract; as such, certain proposals, even if they are accepted, are not guaranteed to go through. Specifically the upgrade process — there is no way to add further validation to enforce a successful upgrade. Upgrade proposals are not always guaranteed to go through, such as when the wasm hash being upgraded to does not exist on chain.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Overview

The Volta contract is a governance contract with multiple authorized users. Authorized users are referred to as owners in this contract.

The owners of the contract can create proposals for various actions and vote on proposals with configurable thresholds.

The following three types of actions can be proposed:

1. **Config** — update the owners and the voting threshold
2. **Invoke** — call functions of other contracts from the current contract
3. **Upgrade** — update the code of the current contract

5.2. Function: `propose`

An owner of the contract can create a new proposal through the function `propose`. The owner must be the signer of the transaction, and this function only supports the two types of actions: `Config` and `Upgrade`.

The initial TTL of all proposals are approximately one week.

Config

If the proposal is for modifying the configuration, a `ConfigInput` must be provided, including the new list of `owners` and the voting `threshold`.

```
#[contracttype]
#[derive(Clone, Debug, PartialEq)]
pub struct ConfigInput {
    pub owners: Vec<Address>,
    pub threshold: u32,
}
```

The length of the owners array cannot be less than `MIN_OWNERS` (2) and cannot contain duplicate

addresses. The `threshold` cannot be less than `MIN_OWNERS` or greater than the number of owners. Lastly, the new configuration cannot be the same as the current configuration.

Upgrade

If the proposal is for upgrading the contract, a WASM hash must be provided.

```
#[contracttype]
#[allow(clippy::large_enum_variant)]
#[derive(Clone)]
pub enum ProposalType {
    // [...]
    Upgrade(BytesN<32>),
}
```

The contract only checks that the input is not all zeros or all `0xFF`. It is necessary to verify off chain that the WASM hash corresponds to a WASM blob that already exists in the ledger and is different from the current contract's WASM hash.

Test coverage

Cases covered

- A proposal of a valid proposal type can be created by an owner.
- Reverts if the `ProposalType` is `Invoke`.
- Reverts if there are duplicate owners.
- Reverts if the new configuration is the same as the current one.
- Reverts if the `threshold` is invalid.
- Reverts if the WASM hash is invalid.
- Reverts if the caller is not an owner.

Cases not covered

- Reverts if the length of the array `owners` is less than `MIN_OWNERS`.

5.3. Function: `invoke`

An owner of the contract can create a new proposal of type `Invoke` through the function `invoke`.

The caller needs to provide the address of the contract to be called, the function name, and the call arguments. If there are secondary calls that require authorization from the current contract,

`auth_entries` also needs to be provided.

```
pub fn invoke(
    env: Env,
    caller: Address,
    contract: Address,
    fn_name: Symbol,
    args: Vec<Val>,
    auth_entries: Vec<InvokerContractAuthEntry>,
) -> Result<(), ContractError> {
```

The address `contract` must exist on the ledger, and the function name cannot be empty.

Test coverage

Cases covered

- A proposal of type `Invoke` can be created by an owner.
- Reverts if the caller is not an owner.

Cases not covered

- Reverts if the address `contract` does not exist on the ledger.
- Reverts if the function name is invalid.

5.4. Function: `revoke_proposal`

The owner who created the proposal can revoke that proposal.

The proposal can only be revoked when its status is `ProposalStatus::Pending`. If the proposal has already been executed, revoked, or automatically invalidated due to contract configuration/code updates, it can no longer be revoked via the function `revoke_proposal`.

Test coverage

Cases covered

- A valid proposal can be revoked by its creator.
- Reverts if the caller is not the proposer.
- Reverts if the proposal is invalid.

Cases not covered

N/A.

5.5. Function: `vote`

An owner can vote on a proposal that is currently in the `Pending` state. Each owner can cast one vote per proposal, which can be `Yes`, `No`, or `Abstain`.

Each time a vote is cast, the function `vote` will update the proposal's status based on the current vote count:

- If the number of `Yes` votes is greater than or equal to the threshold, the proposal status is updated to `Approved`. The proposal is immediately executed and removed from storage.
- If the sum of `Yes` votes and remaining votes is less than the threshold, the proposal is mathematically impossible to reach the threshold, and its status is updated to `Rejected`, after which it is removed from storage.
- Otherwise, the proposal remains in the `Pending` state, and the TTL is extended for approximately one week.

If the proposal is for updating the configuration or code, after execution, the `LastConfigChange` will be updated to the latest proposal ID plus one, invalidating proposals with IDs prior to that number.

Test coverage

Cases covered

- The proposal to update the configuration can be successfully executed through voting.
- The `Invoke` proposal can be successfully executed through voting.
- When the sum of `Yes` votes and remaining votes is greater than or equal to the threshold, the proposal status should be `Pending`.
- When the sum of `Yes` votes and remaining votes is less than the threshold, the proposal is rejected.
- Reverts if the caller is not an owner.

Cases not covered

- The proposal to update the code can be successfully executed through voting.

6. Assessment Results

During our assessment on the scoped Wallet files, we discovered one finding, which was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.