Axelar Stellar

Security Assessment

April 9th, 2025 — Prepared by OtterSec

Andreas Mantzoutas andreas@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Axelar network engaged OtterSec to assess the `amplifier-stellar` program. This assessment was conducted between February 27th and March 19th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a vulnerability where the upgrade process in Stellar upgrader lacks caller authorization, allowing an attacker to front-run the upgrade call, extract its authorization signature, and re-utilize it in a separate transaction to upgrade the contract without migrating (OS-AXT-ADV-00). Additionally, we highlighted an inconsistency in the casting of the contract invocation result, which may result in a panic if the invoked contract returns a value (OS-AXT-ADV-01).

Furthermore, when executing a transfer message, the function emits an event containing raw data, which may exceed Soroban's event size limit of 10KB, resulting in transaction failure and stuck funds (OS-AXT-ADV-02).

We also made suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices (OS-AXT-SUG-01), and advised incorporating more robust validations to avoid potential issues (OS-AXT-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/axelarnetwork/axelar-amplifier-stellar. This audit was performed against commit 458c97d.

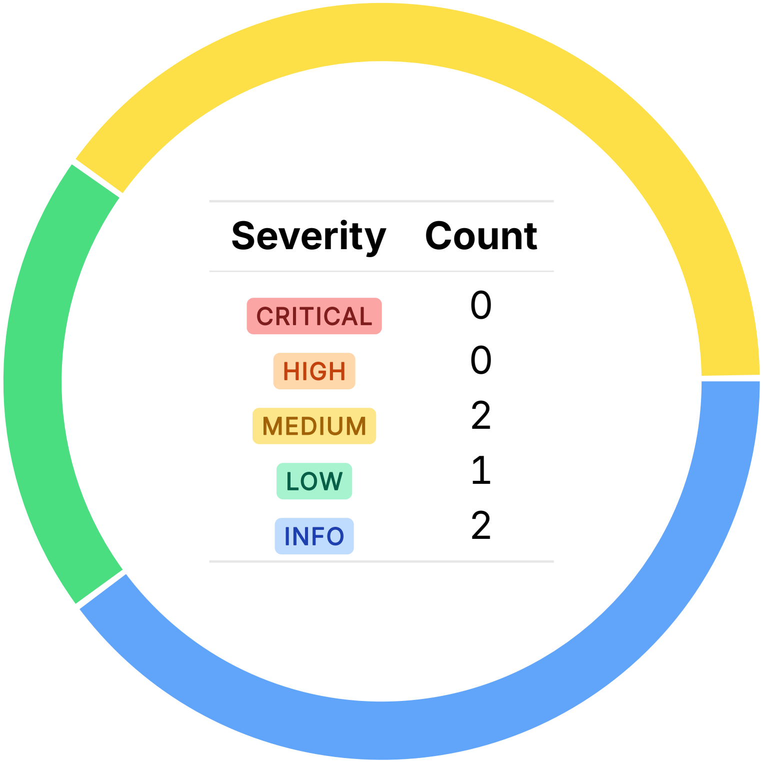
A brief description of the program is as follows:

Name	Description
amplifier-stellar	The repo implements Axelar's cross-chain gateway protocol in Soroban for use on Stellar.

03 — Findings

Overall, we reported 5 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	2
LOW	1
INFO	2

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-AXT-ADV-00	MEDIUM	RESOLVED ✓	The `upgrade` process in `stellar_upgrader` lacks caller authorization, allowing an attacker to front-run the `upgrade` call, extract its authorization signature, and re-utilize it in a separate transaction to upgrade the contract without executing `migrate`.
OS-AXT-ADV-01	MEDIUM	RESOLVED ✓	`execute_contract_with_token` forcefully casts the contract invocation result to `()`, which may result in a panic if the invoked contract returns a value.
OS-AXT-ADV-02	LOW	RESOLVED ✓	`execute_transfer_message` emits an event containing raw data, which may exceed Soroban's event size limit of 10KB, resulting in transaction failure and stuck funds.

Lack of Authorization in Upgrade Process MEDIUM OS-AXT-ADV-00

Description

In `stellar_upgrader`, `upgrade` does not require authorization for the caller. This results in a vulnerability where an attacker may exploit the separation between the contract upgrade and the migration process. Since `upgrade` relies solely on `UpgradableClient` without enforcing strict access control, `upgrade` and `migrate` calls become independent invocations rather than sub-invocations of a single atomic transaction.

>_ *contracts/stellar-upgrader/src/contract.rs* RUST

```rust
 fn upgrade(
    env: Env,
    contract_address: Address,
    new_version: String,
    new_wasm_hash: BytesN<32>,
    migration_data: soroban_sdk::Vec<Val>,
) -> Result<(), ContractError> {
    let contract_client = UpgradableClient::new(&env, &contract_address);
    [...]
    contract_client.upgrade(&new_wasm_hash);
    // The types of the arguments to the migrate function are unknown to this contract, so we
        ↪  need to call it with invoke_contract.
    // The migrate function's return value can be safely cast to () no matter what it really is,
    // because it will panic on failure anyway.
    env.invoke_contract::<()>(&contract_address, &MIGRATE, migration_data);
    [...]
}
```

Consequently, an attacker may frontrun the transaction containing a call to `upgrade`, obtain the `upgrade` call's authorization signature, and utilize that signature to execute a separate `upgrade` transaction, effectively upgrading the contract without invoking `migrate`.

Remediation

Require proper authorization for the caller of `upgrade`, and ensure `upgrade` and `migrate` are executed within a single atomic transaction.

Patch

Fixed in ce4edb7.

Inconsistency in Casting of Contract Invocation Result MEDIUM OS-AXT-ADV-01

Description

`execute_contract_with_token` in `stellar-interchain-token-service` invokes a contract method utilizing `env.invoke_contract::<()>()` . Thus, it always attempts to cast the contract invocation result to `()`, blindly assuming that the invoked function does not return any value. If the target function does return a value, the runtime will panic, potentially resulting in stuck funds.

>_ *contracts/stellar-interchain-token-service/src/contract.rs* RUST

```rust
fn execute_contract_with_token([...]) {
    // Due to limitations of the soroban-sdk, there is no type-safe client for contract
        ↪  execution.
    // The invocation will panic on error, so we can safely cast the return value to `()` and
        ↪  discard it.
    env.invoke_contract::<()>(
        [...]
    );
}
```

Remediation

Cast the result to `Val`, which may hold any return value without resulting in a panic.

Patch

Fixed in 2c48c5a.

Risk of Exceeding Event Size Limit LOW OS-AXT-ADV-02

Description

`execute_transfer_message` emits an `InterchainTransferReceivedEvent`, which includes all transfer details, including the `data` field. However, Soroban imposes a maximum event size limit (around 10KB). While this limit is quite high, if the `data` field is excessively large, it may surpass this threshold, preventing event emission and failing `execute_transfer_message`, resulting in stuck funds.

Remediation

Store only the hash instead of storing the entire `data` payload in the event. This will enable external systems to verify data integrity while ensuring the size remains within limits.

Patch

Fixed in 0ea6545.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-AXT-SUG-00	There are several instances where proper validation is not performed, resulting in potential issues.
OS-AXT-SUG-01	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Missing Validation Logic

OS-AXT-SUG-00

Description

1. `deploy_interchain_token` should assert that either `initial_supply > 0` or `minter.is_some`, otherwise, it will not be possible to utilize the token.
2. `add_minter` currently adds a minter without checking if the address is already a minter. If an address is already a minter, adding them again is unnecessary. Similarly, `remove_minter` removes a minter without checking if they were actually a minter. Before adding or removing, check if the address is already a minter.

   >_ *contracts/stellar-interchain-token/src/contract.rs* RUST

   ```rust
   #[only_owner]
   fn add_minter(env: &Env, minter: Address) {
       storage::set_minter_status(env, minter.clone());
       MinterAddedEvent { minter }.emit(env);
   }

   #[only_owner]
   fn remove_minter(env: &Env, minter: Address) {
       storage::remove_minter_status(env, minter.clone());
       MinterRemovedEvent { minter }.emit(env);
   }
   ```

3. In `flow_limit::add_flow`, setting `flow_limit` too high may introduce a risk of an overflow during the calculation of `max_allowed`, resulting in a denial-of-service scenario. Enforce a maximum cap on `flow_limit` to prevent exceeding a safe value.

   >_ *contracts/stellar-interchain-token-service/src/flow_limit.rs* RUST

   ```rust
   pub fn add_flow([...]) -> Result<(), ContractError> {
       [...]
       let max_allowed = self
           .reverse_flow(env, token_id.clone())
           .checked_add(flow_limit)
           .ok_or(ContractError::FlowAmountOverflow)?;
       // Equivalent to flow_amount + flow - reverse_flow <= flow_limit
       ensure!(new_flow <= max_allowed, ContractError::FlowLimitExceeded);
       [...]
   }
   ```

Remediation

Incorporate the validations mentioned above.

Patch

1. Fixed in `70a71ec`.
2. Fixed in `682bd12`.
3. Fixed in `3c19a97`.

Code Maturity

OS-AXT-SUG-01

Description

1. In `stellar_interchain_token`, the `set_admin` event in `transfer_ownership` emits the new owner's address as both fields of the event after the ownership transfer, instead of emitting the old and new owner. Ensure the event emits the old and new owner for proper logging of ownership changes.

   >_ *contracts/stellar-interchain-token/src/contract.rs* RUST

   ```rust
   fn transfer_ownership(env: &Env, new_owner: Address) {
       interfaces::transfer_ownership::<Self>(env, new_owner.clone());
       // Emit the standard soroban event for setting admin
       TokenEvents::new(env).set_admin(Self::owner(env), new_owner);
   }
   ```

2. Some crates contain un-utilized error codes, such as `InvalidDecimal = 3`, `InvalidTokenName = 4`, and `InvalidTokenSymbol = 5` in `Token`, and `InvalidAddress = 2` in the Gas Service. Remove the unutilized error codes. Also, in `stellar-example::storage`, the `Pause` state variable is unutilized.
3. Limit the external calls to retrieve the token's name and symbol (`token.name` and `token.symbol`) to only occur if the asset is not `XLM` in `token_metadata`, as XLM already has predefined, fixed values for its name and symbol.

Remediation

Implement the suggestions mentioned above.

Patch

1. Fixed in a215f83.
2. Fixed in ad3a708 and f80bdd9.
3. Fixed in 568d832.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.