Axelar Network

Security Assessment

June 11th, 2024 — Prepared by OtterSec

Tuyết Dương tuyet@osec.io

Jessica Clendinen jc0f0@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Axelar Network engaged OtterSec to assess the `cgp-sui` program. This assessment was conducted between April 15th and May 3rd, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 10 findings throughout this audit engagement.

In particular, we identified a vulnerability while estimating the swap balance of one asset from another, where the program calls an incorrect function for retrieving the current balance of the asset being swapped (OS-AXN-ADV-01). We also highlighted the possibility of an incorrect execution order in the transaction process, transferring all balances of a certain coin type to the wrong destination (OS-AXN-ADV-03). Furthermore, ownership transfer calls are not recorded in the Cross-Chain Gateway Protocol, potentially enabling relay attacks (OS-AXN-ADV-02).

We also advised implementing stricter signature validation when the epoch is zero (OS-AXN-SUG-01) and setting a maximum limit on the number of decimal places specified when creating a `CoinInfo` object (OS-AXN-SUG-00). Additionally, we suggested ensuring consistency across implementations of ownership transfer to maintain clarity in handling ownership transfer operations (OS-AXN-SUG-02).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/axelarnetwork/axelar-cgp-sui. This audit was performed against commit 7f5bb51. Additional reviews were performed up to commit 368b2d4. We conducted another follow-up review against commit b13218e.

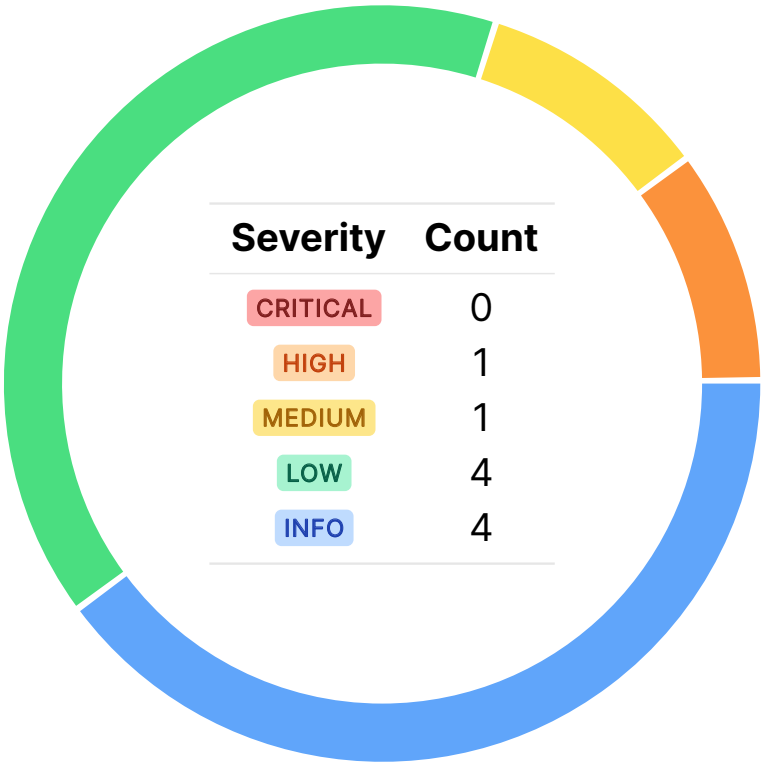
A brief description of the program is as follows:

Name	Description
cgp-sui	An implementation of the Axelar gateway for the Sui blockchain.

03 — Findings

Overall, we reported 10 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	1
MEDIUM	1
LOW	4
INFO	4

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-AXN-ADV-00	HIGH	RESOLVED	`give_coin` currently utilizes `add_flow_out` when transferring coins, which inaccurately tracks the flow of funds during inter-chain transfers.
OS-AXN-ADV-01	MEDIUM	RESOLVED	`estimate` incorrectly calls `get_estimate` for retrieving the current balance of the asset being swapped.
OS-AXN-ADV-02	LOW	RESOLVED	Ownership transfer calls are not recorded in `process_commands` within the Cross-Chain Gateway Protocol, potentially enabling relay attacks.
OS-AXN-ADV-03	LOW	RESOLVED	An incorrect execution order in the transaction process may transfer all balances of a certain coin type to an incorrect destination.
OS-AXN-ADV-04	LOW	RESOLVED	In `validate_signatures`, if the threshold for approval is zero, any number of signatures, even from validators with no weight, will be sufficient.
OS-AXN-ADV-05	LOW	RESOLVED	`set_flow_limit` allows for setting an unlimited flow limit (up to `u64::MAX`), which may result in overflow during calculations and effectively block all transfers.

Utilization of Incorrect Flow Tracking HIGH OS-AXN-ADV-00

Description

The vulnerability concerns the incorrect utilization of flow management in `give_coin` within `CoinManagement`. Specifically, the function currently calls `add_flow_out`, which is intended to track the outflow of tokens, even in situations where it should account for the inflow of tokens when receiving transfers.

>_ *axelar-cgp-sui/move/its/sources/types/coin_management.move* rust

```rust
public(package) fun give_coin<T>(
    self: &mut CoinManagement<T>,
    mut amount: u256,
    clock: &Clock,
    ctx: &mut TxContext,
): Coin<T> {
    amount = amount + self.dust;
    self.dust = amount % self.scaling;
    let sui_amount = (amount / self.scaling as u64);
    self.flow_limit.add_flow_out(sui_amount, clock);
    if (has_capability(self)) {
        self.mint(sui_amount, ctx)
    } else {
        coin::take(self.balance.borrow_mut(), sui_amount, ctx)
    }
}
```

In the current implementation, the function utilizes `self.flow_limit.add_flow_out(sui_amount, clock)` to record the amount of tokens given out. This is inappropriate when the system is receiving tokens through an interchain transfer. Utilizing `add_flow_out` during a reception scenario inaccurately reflects the state of token flow. Instead of tracking tokens that are leaving the system, it should track tokens coming in.

Remediation

Modify `give_coin` to utilize `add_flow_in` instead of `add_flow_out` when receiving tokens.

Patch

Resolved in PR#190.

Incorrect Function Call MEDIUM OS-AXN-ADV-01

Description

`estimate` should estimate the swap balance of one asset (`T2`) from another asset (`T1`). However, instead of utilizing the correct function to retrieve the current balance of the asset being swapped (`T2`), it mistakenly calls `get_estimate`, which retrieves the estimated amounts instead of the actual balances.

>_ *sources/squid/deepbook_v2.move* rust

```
public fun estimate<T1, T2>(self: &mut SwapInfo, pool: &Pool<T1, T2>, clock: &Clock) {
    [...]
    if(has_base) {
        let (amount_left, output) = predict_base_for_quote(
            pool,
            self.coin_bag().get_estimate<T1>(),
            lot_size,
            clock,
        );
        self.coin_bag().store_estimate<T1>(amount_left);
        self.coin_bag().store_estimate<T2>(output);
    }
    [...]
}
```

Since `get_estimate` returns zero (assuming no estimate has been stored), the estimation process will be based on an incorrect or missing balance amount, resulting in inaccurate estimates.

Remediation

Ensure to store the balance of the asset being swapped to `estimate` before calling `get_estimate`.

Patch

Fixed in PR#58.

Absence of Ownership Transfer Recording LOW OS-AXN-ADV-02

Description

There is an issue within `process_commands` in the Cross-Chain Gateway Protocol, specifically in the handling of ownership transfer calls (`SELECTOR_TRANSFER_OPERATORSHIP`). Currently, the function only records approved token transfer calls but does not record calls to transfer ownership. This omission opens up the possibility of relay attacks.

>_ *sources/gateway.move* rust

```
public entry fun process_commands(
    self: &mut Gateway,
    input: vector<u8>
) {
    [...]
    while (i < commands_len) {
        [...]
        else if (cmd_selector == &SELECTOR_TRANSFER_OPERATORSHIP) {
            if (!allow_operatorship_transfer) {
                continue
            };
            allow_operatorship_transfer = false;
            borrow_mut_validators(self).transfer_operatorship(payload);
        } else {
            continue
        };
    };
}
```

Consider a scenario where there are three sets of owners: `A` , `B` , and `C` . Initially, ownership of some assets is transferred from set `A` to set `B` (`setA -> setB`). Later, ownership is further transferred from set `B` to set `C` (`setB -> setC`). Now, if ownership transfer calls are not recorded, it creates a vulnerability when ownership is transferred back from set `C` to set `A` (`setC -> setA`).

Remediation

Enhance `process_commands` to record ownership transfer calls (`SELECTOR_TRANSFER_OPERATORSHIP`).

Patch

Fixed in PR#28.

Potential Balance Misallocation LOW OS-AXN-ADV-03

Description

In `get_transaction`, the call to `sweep` is not explicitly enforced to occur after the call to `swap`. If the program calls `SWAP_TYPE_SWEEP_DUST` before `SWAP_TYPE_DEEPBOOK_V2`, or if there is no `SWAP_TYPE_DEEPBOOK_V2` call, all balances of the base coin (supposedly remaining coins after the swap) will be stored in the `coin_bag` of the `Squid` router in `sweep`.

>_ *sources/squid/discovery.move* rust

```
public fun get_transaction(squid: &Squid, its: &ITS, payload: vector<u8>): Transaction {
    [...]
    while(i < vector::length(&swap_data)) {
        [...]
        vector::push_back(
            &mut move_calls,
            if (swap_type == SWAP_TYPE_SWEEP_DUST) {
                sweep_dust::get_swap_move_call(package_id, bcs, swap_info_arg, squid_arg)
            } else {
                assert!(swap_type == SWAP_TYPE_DEEPBOOK_V2, EInvalidSwapType);
                deepbook_v2::get_swap_move_call(package_id, bcs, swap_info_arg)
            },
        );
        i = i + 1;
    };
}
```

As a result, if `sweep` is called before `swap`, all balances of coin `T1` will be moved to the `coin_bag` of the `Squid` router, and the source or destination address will not receive any coins. `finalize` contains a safeguard to ensure that the balance value (`balance.value()`) is greater than or equal to `self.min_out`. However, this safeguard will not be effective if `min_out` is set to zero.

Remediation

Enforce the correct order of execution between the `swap` and `sweep` calls in `get_transaction`.

Patch

Fixed in PR#28.

Flaw in Signature Approval LOW OS-AXN-ADV-04

Description

The threshold represents the minimum combined weight required from validators to approve a message. Each validator in the signer set has an associated weight, which signifies its voting power or influence. Thus, in `validate_signatures`, if the threshold is zero, it means any number of signatures, regardless of the validators' weights, will be sufficient for approval.

>_ *axelar_gateway/sources/auth.move* rust

```
fun validate_signatures(
    message: vector<u8>,
    signers: &WeightedSigners,
    signatures: &vector<Signature>,
) {
    [...]
    while (i < signatures_length) {
        [...]
        total_weight = total_weight + signers.signers()[signer_index].weight()
        if (total_weight >= threshold) {
            return
        };
        [...]
    }
    abort ELowSignaturesWeight
}
```

This effectively bypasses the intended security mechanism of requiring a minimum level of consensus from validators with voting power. A threshold greater than zero ensures that only a certain level of combined weight from validators is sufficient for approval. This helps prevent unauthorized modifications or actions even if some signatures are compromised.

Remediation

Ensure the threshold is greater than zero within `validate_signatures`.

Patch

Fixed in PR#57.

Possibility of Overflow Due to Uncapped Flow Limit LOW OS-AXN-ADV-05

Description

`set_flow_limit` allows setting the flow limit to any value, including the maximum value that may be stored in a `u64`. This is problematic because, with such a high limit, an overflow may occur in the assertion condition inside `add_flow_out`, which will result in the failure of the assertion check, preventing transfers entirely. While the current system does not actively support setting arbitrary flow limits, a cap should be imposed nonetheless.

>_ axelar-cgp-sui/move/its/sources/types /flow_limit.move rust

```
public(package) fun set_flow_limit(self: &mut FlowLimit, flow_limit: u64) {
    self.flow_limit = flow_limit;
}
```

Remediation

Enforce a reasonable upper bound on the `flow_limit`.

Patch

Resolved in PR#190.

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-AXN-SUG-00	The program allows arbitrary decimal values in `coin_info` creation, which may bypass the `DECIMALS_CAP` limit, resulting in incorrect scaling.
OS-AXN-SUG-01	When the epoch is zero, `validate_proof` immediately returns true without performing any signature validation.
OS-AXN-SUG-02	Handling ownership transfers in two different methods results in inconsistencies.
OS-AXN-SUG-03	`transfer::its_transfer` utilizes an incorrect transfer type in the assertion.

Failure to Restrict Decimal Value Limit

OS-AXN-SUG-00

Description

In `coin_info`, it is possible for anyone to create a `CoinInfo` object utilizing `from_info`, which allows arbitrary decimal values. `from_info` allows users to specify the coin's decimal precision. This value is important because it determines how many fractional parts the coin may have. Due to a lack of restriction on the number of decimals, users may create a coin with extremely large decimal precision and subsequently call `register_coin`. This will result in incorrect scaling of values across the system.

>_ *move/its/sources/types/coin_info.move* rust

```rust
/// Create a new coin info from the given name, symbol and decimals.
public fun from_info<T>(
    name: String,
    symbol: ascii::String,
    decimals: u8,
    remote_decimals: u8,
): CoinInfo<T> {
    CoinInfo {
        name,
        symbol,
        decimals,
        remote_decimals,
        metadata: option::none(),
    }
}
```

Remediation

Verify the decimals specified when creating a `CoinInfo` object.

Unconditional Validation During Epoch Zero OS-AXN-SUG-01

Description

`validators::validate_proof` returns true for any proof validation without performing signature validation when the epoch is zero, allowing the setting of the first operator set. While this is necessary, returning true unconditionally for any proof validation during epoch zero poses a security risk. In particular, `SELECTOR_APPROVE_CONTRACT_CALL` commands without any signatures are also accepted during epoch zero. Thus, approval requests for contract calls may be processed without any validation, resulting in unauthorized contract executions.

>_ *sources/validators.move* rust

```rust
public(package) fun validate_proof(
    validators: &AxelarValidators,
    approval_hash: vector<u8>,
    proof: vector<u8>
): bool {
    let epoch = epoch(validators);
    // Allow the validators to validate any proof before the first set of operators is set (so
        ↪  that they can be rotated).
    if (epoch == 0) {
        return true
    };
    [...]
}
```

Remediation

Restrict unconditional acceptance to only the `SELECTOR_TRANSFER_OPERATORSHIP` command during epoch zero. This ensures that only the transfer of `operatorship` commands is allowed without validation during the initial setup phase. Other approval commands, such as contract call approvals, should only be accepted once the operator set is established and proper signature validation is performed.

Patch

This check has been removed.

Inconsistency in Ownership Transfer Handling OS-AXN-SUG-02

Description

In `validators::transfer_operatorship`, when a new ownership transfer occurs, the existing operator hash is removed if it already exists. However, in the corresponding Solidity code in the Cross-Chain Gateway Protocol, `AxelarAuthWeighted::_transferOperatorship` reverts with a `DuplicateOperators` error if the operator hash already exists.

Remediation

Ensure consistency across implementations and maintain clarity in handling ownership transfer operations.

Patch

Adding duplicate signers now reverts in Gateway V2 because it is not possible to add the same hash via `table::add`.

Incorrect Transfer Types Check

OS-AXN-SUG-03

Description

There is an error in the assertion within `transfer::its_transfer`. `its_transfer` is specifically designed to handle transfers of `ITS` tokens. Since it deals with `ITS` tokens, the expected swap type in the data should be `SWAP_TYPE_ITS_TRANSFER` (defined as a constant three). However, the current code asserts for `SWAP_TYPE_SUI_TRANSFER` (defined as a constant two). This means the function is mistakenly checking for a Sui transfer type even though it's performing an `ITS` transfer.

>_ sources/squid/transfers.move rust

```rust
public fun its_transfer<T>(swap_info: &mut SwapInfo, its: &mut ITS, ctx: &mut TxContext) {
    let data = swap_info.get_data_swapping();
    if (data.length() == 0) return;
    let mut bcs = bcs::new(data)

    assert!(bcs.peel_u8() == SWAP_TYPE_SUI_TRANSFER, EWrongSwapType);
    [...]
}
```

Remediation

Replace `SWAP_TYPE_SUI_TRANSFER` with `SWAP_TYPE_ITS_TRANSFER` in the assertion.

Patch

Fixed in 27fcf68.

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 the other 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.