Sui Bridge

Security Assessment

April 17th, 2024 — Prepared by OtterSec

Tuyết Dương tuyet@osec.io

Jessica Clendinen jc0f0@osec.io

Table of Contents

01 — Executive Summary

Overview

Mysten Labs engaged OtterSec to assess the `suiBridgeV1` program. This assessment was conducted between March 4th and April 9th, 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 high-risk vulnerability involving the possibility of invoking the registration function with an existing public key, failing to create a new committee despite sufficient stake, due to the insertion of duplicate keys in the data structure (OS-SBG-ADV-00). Additionally, we highlighted a mismatch in the order of validators between the input blocklist and the committee's member list, resulting in failed validation due to incorrect indexing (OS-SBG-ADV-01). Furthermore, sending tokens not in a valid route to a target chain is possible, restricting their retrieval from the target chain in the Move bridge system (OS-SBG-ADV-02).

We also advised against redundant storage of system messages (OS-SBG-SUG-00). Furthermore, we suggested implementing more stringent authorization within the system (OS-SBG-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/MystenLabs/sui. This audit was performed against commits 41d4375, cd1e316 and b6c439f.

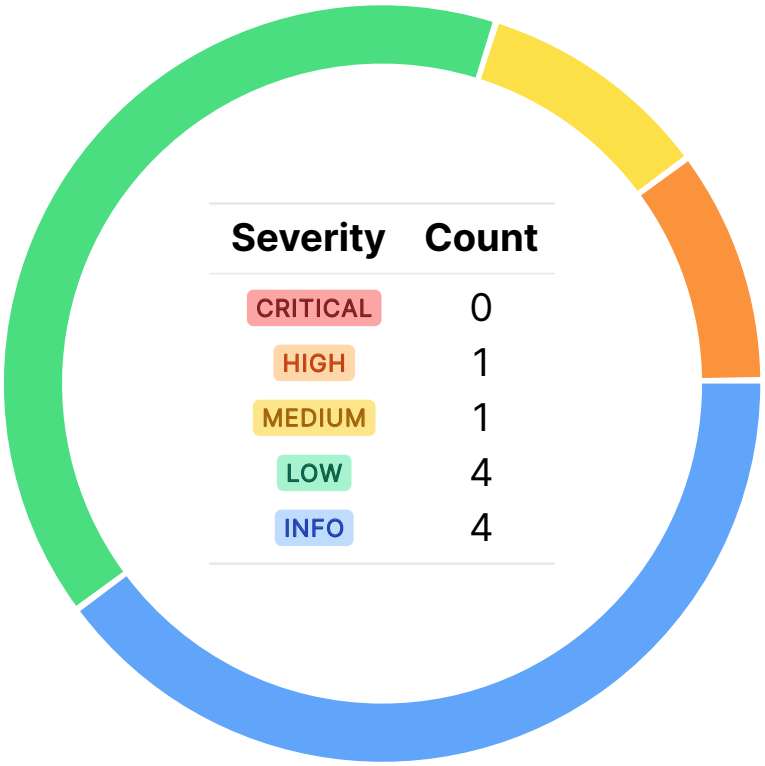
A brief description of the programs is as follows:

Name	Description
suiBridgeV1	The bridge module enables token transfers between blockchain networks, ensuring their security and integrity. It incorporates features such as replay protection, committee-based approval mechanisms, and emergency pause functionalities.

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-SBG-ADV-00	HIGH	RESOLVED	`register` may be invoked with an existing public key, failing to create a new committee despite sufficient stake due to the insertion of duplicate keys in the data structure.
OS-SBG-ADV-01	MEDIUM	RESOLVED	The mismatch in the order of validators between the input `blocklist` and the committee's member list results in failed validation due to incorrect indexing.
OS-SBG-ADV-02	LOW	RESOLVED	It is possible to send tokens to a target chain that is not on a valid route, restricting their retrieval from the target chain in the Move Bridge system.
OS-SBG-ADV-03	LOW	RESOLVED	Users may create a token transfer payload of less than 64 bytes in a Move message by specifying an invalid target address.
OS-SBG-ADV-04	LOW	RESOLVED	`verifySignatures` skips duplicate signatures and unregistered committee members.
OS-SBG-ADV-05	LOW	RESOLVED	Conflicting stake percentages between the Move bridge and the Solidity bridge hinder the creation of a committee for the Solidity bridge, as the minimum stake requirements differ.

Inability To End An Epoch HIGH OS-SBG-ADV-00

Description

`register` allows a validator to register with a used public key (`bridge_pubkey_bytes`). When `try_create_next_committee` is called at the end of epoch, the function creates a new committee based on the registrations stored in `member_registrations`. When attempting to insert the members into the `new_members` mapping utilizing `vec_map::insert`, the insertion will fail if the public key already exists in the mapping. This prevents the system from creating a new committee even if enough stake is available. As a result, the committee is not updated properly, and an end of epoch would fail to attempt to create the committee.

>_ *packages/bridge/sources/committee.move* rust

```rust
if (vector::contains(&validators, &registration.sui_address)) {
    let stake_amount = sui_system::validator_stake_amount(system_state,
        ↪  registration.sui_address);
    let voting_power = ((stake_amount as u128) * 10000) / total_stake_amount;
    total_member_stake = total_member_stake + (stake_amount as u128);
    let member = CommitteeMember {
        sui_address: registration.sui_address,
        bridge_pubkey_bytes: registration.bridge_pubkey_bytes,
        voting_power: (voting_power as u64),
        http_rest_url: registration.http_rest_url,
        blocklisted: false,
    };
    vec_map::insert(&mut new_members, registration.bridge_pubkey_bytes, member)
};
```

Remediation

Restrict the validator from registering with an existing public key.

Patch

Fixed in PR #16909.

Blocklist Validation Order Mismatch MEDIUM OS-SBG-ADV-01

Description

In `committee::execute_blocklist`, the `member_idx` variable is not reset to zero at the beginning of each iteration of the outer `while` loop. Thus, if a `blocklist` contains Ethereum addresses in a different order than the order of committee members stored in `self.members`, the function may fail to find the corresponding committee member even though the member is present in the list.

>_ packages/bridge/sources/committee.move rust

```
public(friend) fun execute_blocklist(self: &mut BridgeCommittee, blocklist: Blocklist) {
    [...]
    while (list_idx < list_len) {
        [...]
        while (member_idx < vec_map::size(&self.members)) {
            [...]
            member_idx = member_idx + 1;
        };
        assert!(found, EValidatorBlocklistContainsUnknownKey);
        list_idx = list_idx + 1;
    };[...]
}
```

If the order of Ethereum addresses in the `blocklist` does not match the order of committee members stored in `self.members`, incorrect behavior ensues. It may abort the function due to failing assertions, indicating that a validator in the `blocklist` was not discovered in `self.members`, even though it exists.

Proof of Concept

Taking `test_execute_blocklist` as an example:

1. Initially, `self.members` contains two members in the following order:
   - Member 1: Ethereum Address = `eth_address0`.
   - Member 2: Ethereum Address = `eth_address1`.
2. Now, suppose the `blocklist` vector contains Ethereum addresses in the opposite order (`vector[eth_address1, eth_address0]`).
3. During the execution of the `execute_blocklist` function, the outer `while` loop iterates over each Ethereum address in the blocklist.

4. For the first Ethereum address (`eth_address1`) in the blocklist:
    - The function begins searching for a matching member in `self.members` starting from the current `member_idx` .
    - Since `member_idx` was not reset to zero at the beginning of the loop, it retains its previous value (index where the last search ended).
    - If `member_idx` is not pointing to the correct index where `eth_address1` is stored, the function will not find a match, and the assertion `assert!(found, EValidatorBlocklistContainsUnknownKey)` will fail.
5. Similarly, for the second Ethereum address (`eth_address0`) in the blocklist, the function continues the search from the incorrect `member_idx` , potentially failing to find the correct member.

Remediation

Reset the `member_idx` variable to zero at the beginning of each iteration of the outer `while` loop. This ensures that the search for a committee member starts from the beginning of the member list for each Ethereum address in the `blocklist` , regardless of its order.

Patch

Fixed in PR #16955.

Inadvertent Locking Of Tokens In Incorrect Chain LOW OS-SBG-ADV-02

Description

`bridge::send_token` permits users to specify a `target_chain` parameter, indicating the blockchain network to which they wish to send their tokens. However, it neglects to verify whether the specified target chain is part of a valid route in the Move Bridge system.

>_ *packages/bridge/sources/bridge.move* rust

```
public fun send_token<T>(
    self: &mut Bridge,
    target_chain: u8,
    target_address: vector<u8>,
    token: Coin<T>,
    ctx: &mut TxContext
) {
    [...]
    // create bridge message
    let message = message::create_token_bridge_message(
        inner.chain_id,
        bridge_seq_num,
        address::to_bytes(tx_context::sender(ctx)),
        target_chain,
        target_address,
        token_id,
        amount,
)
};
```

Without validating the target chain against a predefined route map or routing logic, users risk inadvertently selecting a target chain incapable of processing token transfers or lacking interoperability with the source chain. This effectively locks the tokens on that chain, preventing them from being claimed by their intended recipients.

Remediation

Implement validation mechanisms within `send_token` to ensure that the target chain specified by users is part of a valid route in the Move Bridge system.

Patch

Fixed in PR #16960.

Payload Size Limitation LOW OS-SBG-ADV-03

Description

Users may create a token transfer payload with less than 64 bytes, resulting in an invalid target address in the Move message. Consequently, when attempting to claim the tokens on Ethereum, the invalid target address would prevent the successful execution of the transaction, thereby rendering the tokens unclaimable on the Ethereum chain.

Remediation

Implement proper validation checks on the payload size and content within the Move message to ensure it satisfies the required criteria.

Patch

Fixed in PR #17139.

Signature Handling Inconsistency LOW OS-SBG-ADV-04

Description

In the Solidity bridge, `verifySignatures` skips duplicate signatures and unregistered signers utilizing the `continue` statement. This method deviates from the stricter validation approach adopted by the Move bridge code, which aborts on such conditions.

>_ *evm/contracts/BridgeCommittee.sol* rust

```
function verifySignatures(bytes[] memory signatures, BridgeMessage.Message memory message)
    public
    view
    override
{
    [...]
    // skip if signer is block listed or has no stake
    if (blocklist[signer] || committeeStake[signer] == 0) continue;
    [...]
    else {
        // skip if duplicate signature
        continue;
    }
    [...]
}
```

Remediation

Adopt stricter validation within `verifySignatures`, following the approach used in the Move bridge code.

Patch

Fixed in PR #16934.

Discrepancy In Minimum Stake Requirement LOW OS-SBG-ADV-05

Description

The Move bridge requires a minimum stake participation of 60%, whereas the Solidity bridge initialization mandates a hard-coded total stake of 100%. This inconsistency creates a scenario where it may not be possible to create a committee for the Solidity bridge, as the total stake required by the Solidity bridge exceeds the minimum stake participation set by the Move bridge.

>_ *evm/contracts/BridgeCommittee.sol* rust

```
    require(
            committeeStake[committee[i]] == 0, "BridgeCommittee: Duplicate committee member"
        );
        committeeStake[committee[i]] = stake[i];
        committeeIndex[committee[i]] = uint8(i);
        totalStake += stake[i];
    }

    require(totalStake == 10000, "BridgeCommittee: Total stake must be 10000"); // 10000 == 100%
}
```

Remediation

Harmonize the stake requirements between the Move bridge and the Solidity bridge.

Patch

Fixed in PR #16449.

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-SBG-SUG-00	`approve_bridge_message` redundantly stores system messages.
OS-SBG-SUG-01	There is unauthorized manipulation of transfer records within the system, disrupting the token-claiming process for a certain period.
OS-SBG-SUG-02	`add_new_token` allows a `notional_value` of zero, which was impossible before due to hard-coded values.
OS-SBG-SUG-03	Incorrect error information.

Redundant Message Storage OS-SBG-SUG-00

Description

`bridge::approve_bridge_message` stores both token messages and system messages in the `BridgeRecord` data structure. However, storing system messages is redundant since they execute immediately after verification in `execute_system_message`. This redundant storage adds unnecessary complexity to the code base, increasing the risk of confusion or errors during message retrieval or processing.

>_ *packages/bridge/sources/bridge.move* rust

```
public fun approve_bridge_message(
    self: &mut Bridge,
    message: BridgeMessage,
    signatures: vector<vector<u8>>,
) {
    [...]
    else{
    [...]
        linked_table::push_back(&mut inner.bridge_records, key, BridgeRecord {
            message,
            verified_signatures: some(signatures),
            claimed: false
        });
    };
    emit(TokenTransferApproved { message_key: key });
}
```

Remediation

Update `approve_bridge_message` to store only token messages in the `BridgeRecord` data structure. System messages execute directly within `execute_system_message` without being stored in `BridgeRecord`.

Patch

Fixed in PR #17579.

Unauthorized Transfer Record Manipulation OS-SBG-SUG-01

Description

`limiter::check_and_record_sending_transfer` permits the addition of limiter records without executing an actual transfer, enabling invocation of this function with arbitrary input parameters, such as a large amount, without affecting any token transfer. This creates a loophole where transfer records may be manipulated without corresponding token movements, disrupting the system's normal operation by hindering legitimate users from claiming tokens or conducting transfers within expected limits.

Remediation

Limit the invocation of `check_and_record_sending_transfer` to authorized modules or components.

Patch

Fixed in PR #17579.

Lack Of Notional Value Validation OS-SBG-SUG-02

Description

In `treasury::add_new_token`, there is currently no check to ensure that the `notional_value` of the token being added is greater than zero. This means setting the `notional_value` to zero is possible. This was impossible in the older version due to a hard-coded value.

>_ *packages/bridge/sources/treasury.move* rust

```
public(package) fun add_new_token(self: &mut BridgeTreasury, token_name: String, token_id:u8,
    ↪  native_token: bool, notional_value: u64) {
    if (!native_token){
        [...]
        vec_map::insert(&mut self.supported_tokens, type_name, BridgeTokenMetadata{
            id: token_id,
            decimal_multiplier,
            notional_value,
            native_token
        });
        vec_map::insert(&mut self.id_token_type_map, token_id, type_name);
    [...]
}
```

Remediation

Add a check in `add_new_token` which ensures `notional_value > 0`.

Patch

Fixed in PR #17573.

Incorrect Error Information

OS-SBG-SUG-03

Description

When `handle_add_tokens_on_sui` encounters a situation where the `chain_id` does not correspond to a Sui chain, it returns an error. However, this error message is incorrect as it mentions `handle_add_tokens_on_evm` instead of `handle_add_tokens_on_sui`.

>_ *sui-bridge/src/server/mod.rs* rust

```rust
async fn handle_add_tokens_on_sui(
    [...]
) -> Result<Json<SignedBridgeAction>, BridgeError> {
    [...]
    if !chain_id.is_sui_chain() {
        return Err(BridgeError::InvalidBridgeClientRequest(
            "handle_add_tokens_on_evm only expects Sui chain id".to_string(),
        ));
    }
    [...]
}
```

Remediation

Modify the error message to mention `handle_add_tokens_on_sui` instead of `handle_add_tokens_on_evm`.

Patch

Fixed in PR #17573.

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.