Wormhole NTT

Security Assessment

April 21st, 2025 — Prepared by OtterSec

Ajay Shankar Kunapareddy d1r3wolf@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Wormhole Foundation engaged OtterSec to assess the `sui-ntt` program. This assessment was conducted for a total of 3 weeks between April 7th and April 18th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we identified a high-risk denial-of-service vulnerability in the receive message instruction in wormhole transceiver, where a malicious transceiver message with a re-utilized id may preempt a legitimate transfer, blocking PDA creation for the original message (OS-WPH-ADV-00).

We also provided recommendations to ensure adherence to coding best practices (OS-WPH-SUG-00) and suggested modifying the codebase for improved functionality, and to mitigate potential security issues (OS-WPH-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/wormholelabs-xyz/native-token-transfers. This audit was performed against commit 690a694.

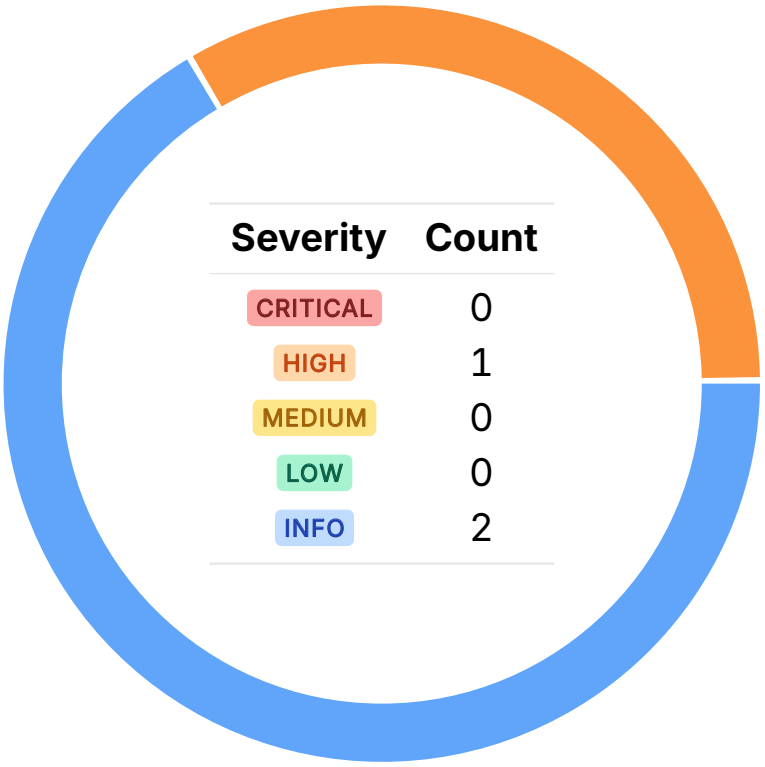
A brief description of the program is as follows:

Name	Description
sui-ntt	Sui NTT enables secure, modular cross-chain transfers of native tokens on the Sui blockchain. It separates token transfer logic (NTT Manager) from message transport (Transceiver, e.g., Wormhole) using a shared interface (ntt-common). Messages are passed via permissioned structures in programmable transaction blocks.

02 — Findings

Overall, we reported 3 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	0
LOW	0
INFO	2

03 — 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-WPH-ADV-00	HIGH	RESOLVED	There is a denial-of-service vulnerability in `receive_message` instruction where a malicious transceiver message with a re-utilized ID may preempt a legitimate transfer, blocking PDA creation for the original message.

DOS due to Transceiver Message Collision HIGH OS-WPH-ADV-00

Description

In the current implementation on Sui, any NTT manager may send messages via a transceiver to any destination manager, implying the message IDs may be arbitrary. However, Solana's `receive_message` instruction for Wormhole NTT utilizes `chain_id` and `message.id` as seeds for the `ValidatedTransceiverMessage` PDA to store validated transceiver messages, since there can only be one NTT Manager per transceiver on solana.

Thus, an attacker on sui may submit a malicious message with the same id as a legitimate transfer before it arrives onto solana, initializing the PDA pre-emptively. This results in a denial of service for the real transfer, as the PDA would already exist.

>_ *src/transceivers/wormhole/instructions/receive_message.rs* RUST

```rust
#[account(
    init,
    payer = payer,
    space = 8 + ValidatedTransceiverMessage::
    <TransceiverMessageData<NativeTokenTransfer<Payload>>>::INIT_SPACE,
    seeds = [
        ValidatedTransceiverMessage::
        <TransceiverMessageData<NativeTokenTransfer<Payload>>>::SEED_PREFIX,
        vaa.emitter_chain().to_be_bytes().as_ref(),
        vaa.message().ntt_manager_payload.id.as_ref(),
    ],
    bump,
)]
pub transceiver_message:
    Account<'info, ValidatedTransceiverMessage<NativeTokenTransfer<Payload>>>,
```

Remediation

Re-evaluate the logic to confirm whether transceivers are intended to support messages from multiple NTT managers on Solana as well.

Patch

Resolved on Sui in 3ff72c2.

04 — 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-WPH-SUG-00	Suggestions regarding ensuring adherence to coding best practices.
OS-WPH-SUG-01	Recommendation for modifying the codebase for improved functionality, and to mitigate potential security issues.

Code Maturity OS-WPH-SUG-00

Description

1. Currently, in `bytes4::trim_nonzero`, the function comment only describes the behavior of trimming leading zero bytes and aborting if a non-zero byte is encountered. However, the function will also abort if the input length is less than 4. It will be appropriate to update the comment to reflect this constraint.

>_ *ntt_common/sources/datatypes/bytes4.move* RUST

```
/// Trim bytes from the left if they are zero. If any of these bytes
/// are non-zero, abort.
fun trim_nonzero_left(data: &mut vector<u8>) {
    vector::reverse(data);
    let (mut i, n) = (0, vector::length(data) - LEN);
    while (i < n) {
        assert!(vector::pop_back(data) == 0, E_CANNOT_TRIM_NONZERO);
        i = i + 1;
    };
    vector::reverse(data);
}
```

2. Although `ntt::redeem` checks `to_chain`, re-verifying it in `ntt::release_impl` prevents the release process from proceeding with a mismatched destination chain. This low-cost verification improves the system's overall integrity.
3. The `TODO` in `ntt_common::contract_auth` is valid, as the current `contract_auth` mechanism accepts any structure named `Auth`, which is overly generic and lacks fine-grained access control. To enhance security and clarity, it is recommended to utilize more specific and semantically meaningful auth types, such as `NTTSendAuth`.

Remediation

Implement the above-mentioned suggestions.

Code Refactoring OS-WPH-SUG-01

Description

1. `broadcast_id` in `wormhole_transceiver` derives the manager's address from the `Auth` type but accepts a `ManagerState` as an external input without verifying that both originate from the same package. This creates a risk of broadcasting an incorrect `ManagerState` value for a given `manager_address`. Verify that the `manager_state` is indeed owned by the address returned by `get_auth_address<Auth>()`.

>_ *wormhole_transceiver/sources/wormhole_transceiver.move* RUST

```
public fun broadcast_id<CoinType, Auth>(_: &AdminCap, coin_meta: &CoinMetadata<CoinType>,
    ↪  state: &mut State, manager_state: &ManagerState<CoinType>): MessageTicket {
    let mut manager_address_opt: Option<address> =
        ↪  ntt_common::contract_auth::get_auth_address<Auth>();
    let manager_address = option::extract(&mut manager_address_opt);
    let external_address_manager_address =
        ↪  wormhole::external_address::from_address(manager_address);
    [...]
}
```

2. In `wormhole_transceiver_info` and `wormhole_transceiver_registration` modules, `new` is marked `public(package)`, restricting structure creation to internal code. In contrast, `parse` is public, allowing any external module to deserialize and construct these structures from raw bytes. Ensure that proper access control is enforced.
3. `transceiver_registry::next_id` assigns a unique ID to each transceiver, utilizing it as a bit index in a fixed-size 128-bit Bitmap. However, without bounds checking, the ID may exceed 127, resulting in out-of-bounds errors when accessing the bitmap. Add an `assert!(id < 128)` check in `next_id`. This ensures safe utilization of bitmap indices and limits the number of registered transceivers to 128.

>_ *ntt_common/sources/transceiver_registry.move* RUST

```
public fun next_id(registry: &mut TransceiverRegistry): u8 {
    let id = registry.next_id;
    registry.next_id = id + 1;
    id
}
```

4. Utilizing a mutable reference (`&mut Auth`) in `contract_auth::assert_auth_type` enhances security by ensuring exclusive access to the `Auth` object, preventing unintended re-utilization or sharing.

>_ ntt_common/sources/contract_auth.move RUST

```rust
public fun assert_auth_type<Auth>(auth: &Auth): address {
    let maybe_addy = get_auth_address<Auth>();
    if (maybe_addy.is_none()) {
        abort EInvalidAuthType
    };
    *maybe_addy.borrow()
}
```

Remediation

Incorporate the above refactors into the codebase.

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.