Wormhole NTT

Security Assessment

March 28th, 2024 — Prepared by OtterSec

Harrison Green hgarrereyn@osec.io

Robert Chen notdeghost@osec.io

James Wang james.wang@osec.io

Table of Contents

01 — Executive Summary

Overview

Wormhole Foundation engaged OtterSec to assess the `example-native-token-transfers` program. This assessment was conducted between February 13th and March 26th, 2024. 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 high-risk vulnerability that permits replay attacks by reusing previously submitted governance messages, potentially resulting in the unintended execution of instructions (OS-WPH-ADV-00).

We also provided recommendations regarding the absence of verification, which may pose a front-running risk if ownership is not confirmed (OS-WPH-SUG-00), and highlighted potential issues when associated token accounts have different owners, possibly due to phishing attacks or account compromise (OS-WPH-SUG-02). Furthermore, we addressed discrepancies in the calculation of serialization sizes and the lack of considerations for specific data fields (OS-WPH-SUG-03).

Scope

The source code was delivered to us in a Git repository at https://github.com/wormhole-foundation/example-native-token-transfers. This audit was performed against commit 06634b9.

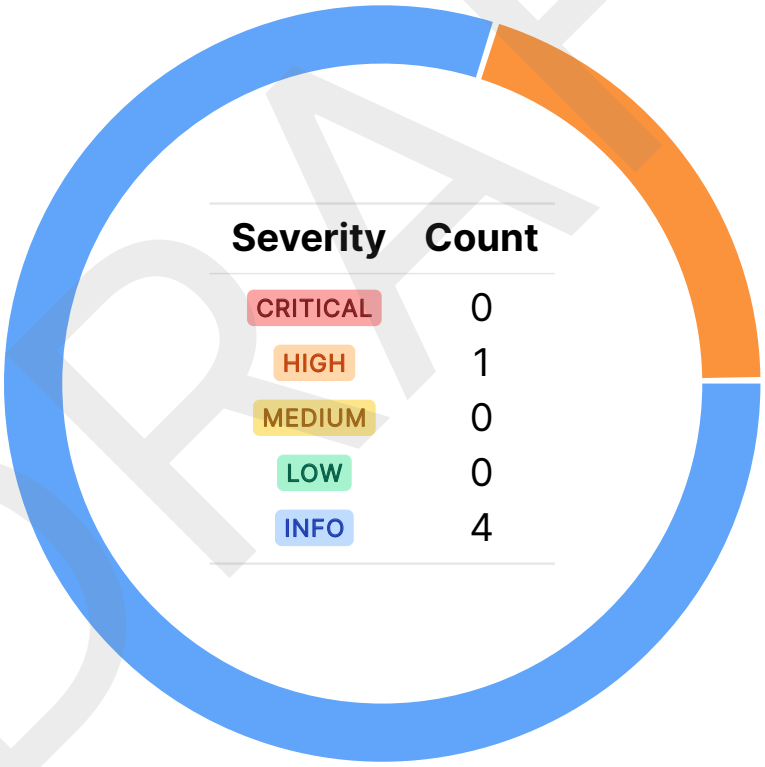
A brief description of the programs is as follows:

Name	Description
example-native-oken-transfers	Wormhole's native token transfers is an open, flexible, and composable framework for transferring tokens across blockchains without liquidity pools. Integrators retain full control over the behavior of their natively transferred tokens on each chain, including the token standard and metadata.

02 — 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	1
MEDIUM	0
LOW	0
INFO	4

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	The vulnerability permits replay attacks by reusing previously submitted governance messages, potentially resulting in the unintended execution of instructions.

Governance Replay Attack HIGH

OS-WPH-ADV-00

Description

The `governance` account within the `Governance` structure may be designated as `config.owner`, indicating potential control by an external entity. Failure to adequately manage or secure this `governance` account may result in the vulnerability of replay attacks on previous `GovernanceMessage` instances.

An attacker may intercept and retain previously executed `GovernanceMessage` instances. Subsequently, upon gaining control of the governance account, they may re-submit these stored messages, effectively replaying the same instructions. This action may result in unintended effects or exploits within the governed program.

Remediation

Ensure that each `GovernanceMessage` instance is unique, so it may not be replayed once executed.

Patch

Fixed in db40415.

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	The issue arises from the absence of verification that the `from` account in the `Transfer` structure is owned by the sender, which may pose a frontrunning risk if ownership is not confirmed.
OS-WPH-SUG-01	In `initialize`, a risk of denial of service exists.
OS-WPH-SUG-02	Token transfers may encounter potential issues when associated token accounts have different owners, possibly due to phishing attacks or account compromise.
OS-WPH-SUG-03	The calculation of serialization sizes shows discrepancies and lacks considerations for specific data fields. This results in inefficient memory usage and the potential for data truncation during serialization.

Transfer Frontrunning OS-WPH-SUG-00

Description

In `transfer`, the `from` account denotes the token account from which tokens are debited during the transfer operation. However, the current implementation lacks an explicit check to verify that the `from` account is owned by the sender. This omission introduces a potential front-running risk.

>_ *example-native-token-transfers/src/instructions/transfer.rs* rust

```rust
pub struct Transfer<'info> {
    [...]
    #[account(
        mut,
        token::mint = mint,
    )]
    pub from: InterfaceAccount<'info, token_interface::TokenAccount>,
    [...]
}
```

If the `from` the sender does not explicitly own the account, there is a possibility that another party may observe the pending transfer and submit their own transaction to transfer tokens from the `from` account before the intended transfer takes place.

Remediation

Ensure that the sender owns the `from` account used in the transfer operation.

Patch

Fixed in 0a87b95.

Risk Of Denial Of Service

OS-WPH-SUG-01

Description

Associated token accounts are created dynamically and linked to a particular `SPL` token mint. Improper handling may allow an attacker to overwhelm a program's initialization (`initialize`) method with requests to create associated token accounts, resulting in a denial of service attack due to excessive initialization overhead.

Remediation

Use `init_if_needed` on associated token accounts to mitigate the risk of denial of service attacks caused by reinitialization.

Discrepancies In Associated Token Account Ownership OS-WPH-SUG-02

Description

The issue involves potential discrepancies between the intended and actual ownership of the associated token account. In situations where the actual ownership of the associated token account does not align with the expected ownership, such as in cases of phishing attacks or unauthorized account access, the current setup may abort the transaction.

Remediation

Ensure explicit handling for `recipient` account in `release_inbound`.

Serialization Size Mismatch OS-WPH-SUG-03

Description

The purpose of `written_size` is to compute the size in bytes required to serialize an instance of the `TransceiverMessage` type. However, the function fails to accurately determine the size of the serialized data, potentially resulting in inefficient heap space usage when pre-allocating memory in `to_vec_payload`.

>_ *ntt-messages/src/transceiver.rs* rust

```rust
impl<E: Transceiver, A: TypePrefixedPayload> Writeable for TransceiverMessage<E, A>
where
    A: MaybeSpace,
{
    fn written_size(&self) -> usize {
        4 // prefix
        + self.source_ntt_manager.len()
        + u16::SIZE.unwrap() // length prefix
        + self.ntt_manager_payload.written_size()
    }
    [...]
}
```

`written_size` fails to account for all fields of the `TransceiverMessage` type, particularly `recipient_ntt_manager` and `transceiver_payload`, resulting in an incomplete calculation of the serialized size. Additionally, the `Readable` implementation for `WormholeTransceiverRegistration` and `WormholeTransceiverInfo` incorrectly specifies the size, as it should include the size of the prefix.

>_ *ntt-messages/src/transceivers/wormhole.rs* rust

```rust
impl Readable for WormholeTransceiverInfo {
    const SIZE: Option<usize> = Some(32 + 1 + 32 + 1);
    [...]
}

impl Readable for WormholeTransceiverRegistration {
    const SIZE: Option<usize> = Some(2 + 32);
    [...]
}
```

Remediation

Ensure that `written_size` accurately calculates the size of the serialized data to prevent inefficient heap space usage. This includes correctly accounting for all fields of the `TransceiverMessage` type, such as `recipient_ntt_manager` and `transceiver_payload`. Additionally, update the `SIZE` constant for `WormholeTransceiverRegistration` and `WormholeTransceiverInfo` to include the size of the prefix.

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.