OtterSec

Wormhole Shims

Security Assessment

March 7th, 2025 — Prepared by OtterSec

Xiang Yin	soreatu@osec.io
Gabriel Ottoboni	ottoboni@osec.io
Kevin Chow	kchow@osec.io
Robert Chen	r@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope **3**

Findings **4**

General Findings **5**

OS-WSH-SUG-00 | Missing Ownership Validation 6

OS-WSH-SUG-01 | Incorrect Usage Of Unsafe Function 7

Appendices

Vulnerability Rating Scale **8**

Procedure **9**

01 — Executive Summary

Overview

Wormhole Labs engaged OtterSec to assess the `wormhole-core-shims` program. This assessment was conducted between February 12th and February 26th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings during this audit engagement.

In particular, we identified instances of unsafe code that may result in undefined behavior (OS-WSH-SUG-01). We also recommended adding a check to validate the owner of the guardian signatures account in the hash verification process (OS-WSH-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/wormholelabs-xyz/wormhole. This audit was performed against commit f69b3ae.

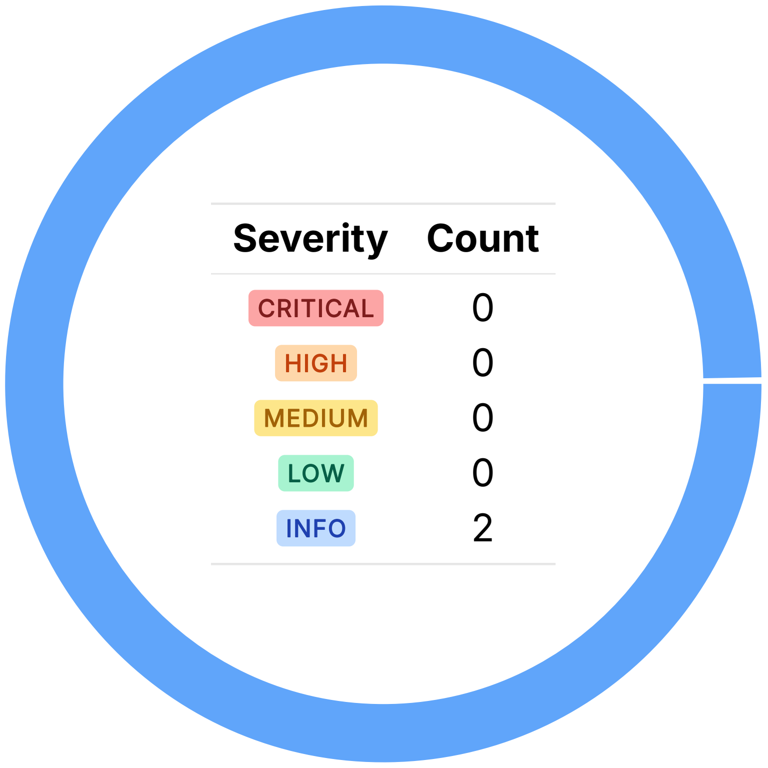
A brief description of the program is as follows:

Name	Description
wormhole-core-shims	The shims optimize the bridge program by reducing the cost of core bridge message emission and verification on Solana without the existing Wormhole core bridge.

03 — Findings

Overall, we reported 2 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	0
LOW	0
INFO	2

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-WSH-SUG-00	`process_verify_hash` does not validate the owner of the `guardian_signatures` account against `VERIFY_VAA_SHIM_PROGRAM_ID`.
OS-WSH-SUG-01	`create_account_reliably` and `process_post_message` call `set_len` without initializing the elements of the modified `Vec`, which may result in undefined behavior.

Missing Ownership Validation

OS-WSH-SUG-00

Description

Currently, `process_verify_hash` does not check whether `guardian_signatures` account is owned by the expected program (`VERIFY_VAA_SHIM_PROGRAM_ID`). This implies that any account that can be deserialized into `GuardianSignatures` may be utilized in the verification process.

Remediation

Modify the function to add an ownership check for the `guardian_signatures` account.

Patch

Resolved in a0ffda4.

Incorrect Usage Of Unsafe Function OS-WSH-SUG-01

Description

`create_account_reliably` and `process_post_message` do not ensure that all the entries of the modified `Vec` were initialized up until the new length before calling `set_len`. According to the function's documentation, this is one of the invariants that must be upheld to avoid undefined behavior.

>_ *wormwhole-core-shims/programs/post-message/src/lib.rs* RUST

```rust
fn process_post_message(accounts: &[AccountInfo]) -> ProgramResult {
    [...]
    unsafe {
        cpi_data.set_len(MAX_CPI_DATA_LEN);
    }
    [...]
```

>_ *wormwhole-core-shims/programs/verify-vaa/src/lib.rs* RUST

```rust
fn create_account_reliably(
    [...]
    unsafe {
        core::ptr::write_bytes(cpi_data.as_mut_ptr(), 0, 4);
        cpi_data.set_len(12);
    }
    [...]
    unsafe {
        cpi_data.set_len(MAX_CPI_DATA_LEN);
    }
    [...]
```

Remediation

Ensure every entry up until the new length is initialized before calling `set_len`.

Patch

Partially resolved in 32cb65d.

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.