Wormhole Sui

Audit

Presented by:

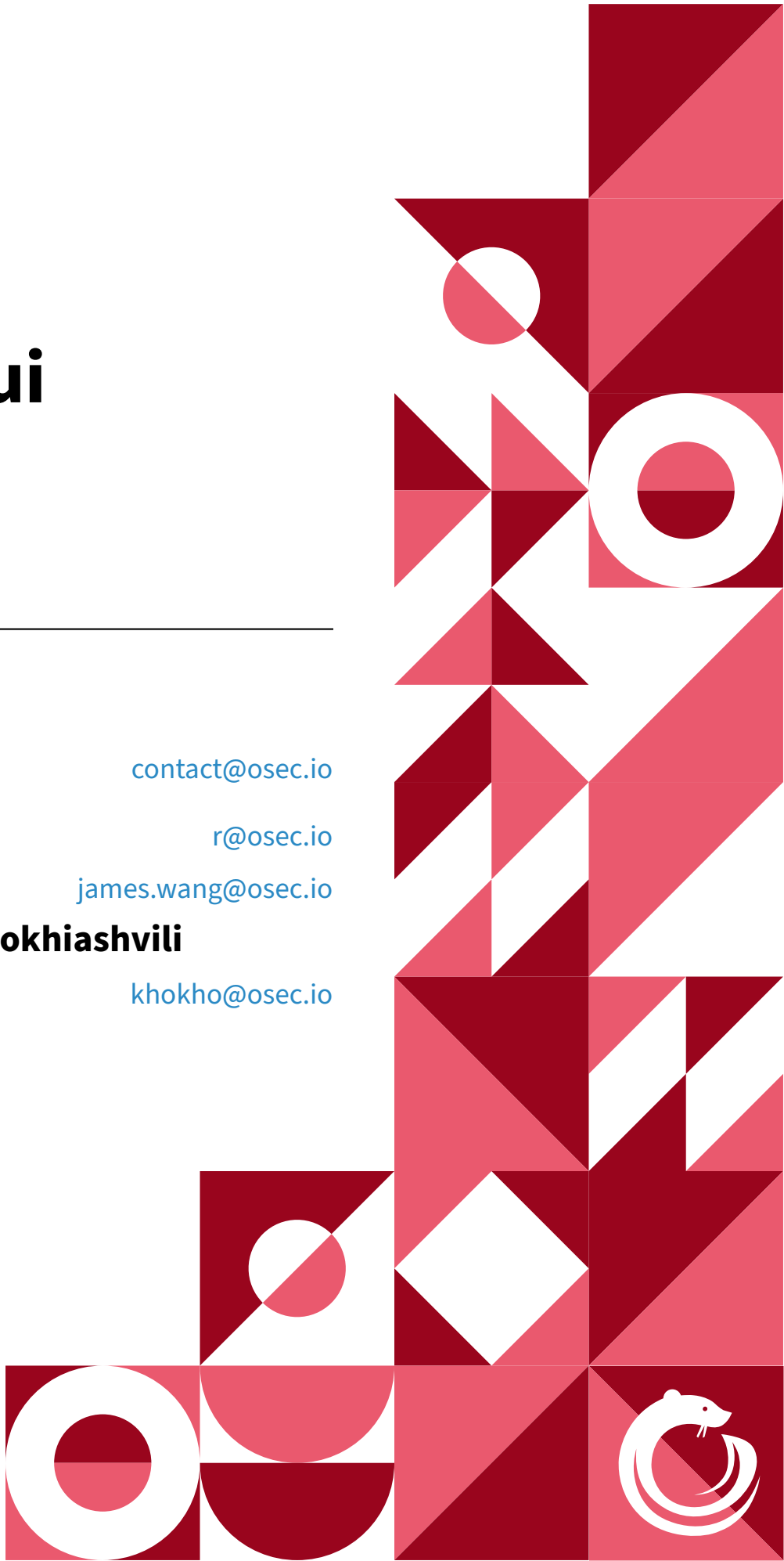
OtterSec contact@osec.io

Robert Chen r@osec.io

James Wang james.wang@osec.io

Aleksandre Khokhiashvili

khokho@osec.io



Contents

01 | Executive Summary

Overview

Wormhole Foundation engaged OtterSec to perform an assessment of the `wormhole`, `token_bridge`, and `watcher` programs. This assessment was conducted between April 12th and May 5th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 5 findings total.

In particular, we addressed issues regarding a weakness in the upgrade commit procedure that may lead to state corruptions (OS-WHS-ADV-00) and watcher mishandling of messages that may lead to misinterpretation (OS-WHS-ADV-01).

We also made recommendations around additional checks to prevent governance mistakes (OS-WHS-SUG-00).

Scope

The source code was delivered to us in a git repository at github.com/wormhole-foundation/wormhole. This audit was performed against commit f62bd11 for `wormhole` and `token_bridge`, and 40a638d for `watcher`.

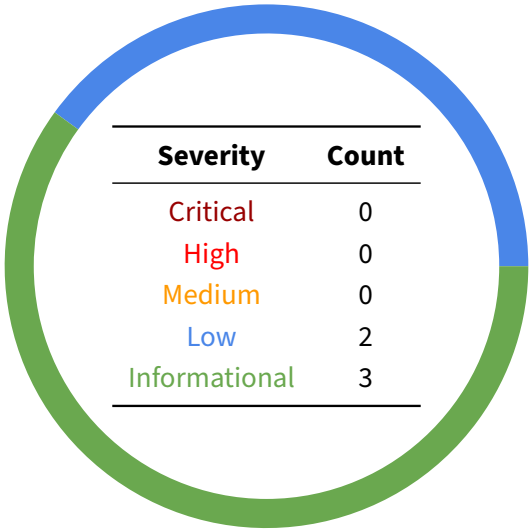
A brief description of the programs is as follows.

Name	Description
wormhole	`wormhole` handles validating and emitting cross-chain VAA messages. Additional upgrade functionalities are also implemented for the Sui version of Wormhole.
token_bridge	`token_bridge` is responsible for attesting, registering and bridging tokens across chains. The `wormhole` module is used to send cross-chain messages. Upgrade functionalities are also included for the `token_bridge` module.
watcher	`watcher` receives messages from the wormhole contract and emits them as observations. `watcher` is also responsible for querying block height and handling re-observation requests.

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 don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	0
Low	2
Informational	3

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-WHS-ADV-00	Low	Resolved	Users may perform actions between the package `upgrade` and `migrate`.
OS-WHS-ADV-01	Low	Resolved	Checking against errors by searching for substrings in marshalled messages may result in misinterpretation.

OS-WHS-ADV-00 [low] | Non Atomic Upgrade And Migrate

Description

The original implementation of `wormhole` and `token_bridge` upgrade consists of two steps.

1. Upgrade the package through the native Sui upgrade command.
2. Custom migration logic facilitates the switch from old to new modules.

Due to a circular dependency between `storageId` of the newly deployed package and the hash of `BatchTransaction` that includes the upgrade transaction, it is generally not possible to call functions on new modules within the same `BatchTransaction` where the upgrade command is included.

This limitation forces `wormhole` and `token_bridge`'s `upgrade` and `migrate` to be performed in two different transactions and exposes a window where other transactions may be called on both new and old packages after `upgrade` occurs but before `migrate` is completed.

Remediation

The Wormhole team has decided to rework the upgrade flow to:

1. Minimize public functions that should be used by external integrator packages. This simplifies version control logic by removing modular upgrades and opting for a singular version across all modules.
2. Add a version toggle that would get activated upon calling `migrate`. This ensures that before `migrate` occurs, only functions from an old package with APIs compatible with the pre-migrate state may be called.

Patch

Resolved in 760db3c.

OS-WHS-ADV-01 [low] | Handle Watcher Message Properly

Description

The current `watcher` implementation is directly checking marshalled return messages for an `"error"` string to decide whether an error occurred.

node/pkg/watchers/sui/watcher.go GO

```go
if strings.Contains(string(body), "error") {
    logger.Error("Failed to get events for re-observation request",
    ↪  zap.String("Result", string(body)))
    continue
}
```

Successful messages that contain `"error"` in the body will incorrectly be caught and misinterpreted as an actual error.

Remediation

Properly unmarshal query results and correctly check whether field `"error"` exists in the response JSON object. One possible way to do this is by following the pattern found in the process account subscription data function of the Solana watcher found here, which correctly implements the same behavior.

Patch

Resolved in 3eefb74.

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 antipatterns and could lead to security issues in the future.

ID	Description
OS-WHS-SUG-00	Add checks to governance action updating `guardian_set` to prevent accidental additions of the same `guardian` twice.
OS-WHS-SUG-01	Make internal state-changing functions `friend` only to minimize the attack surface.
OS-WHS-SUG-02	Fix incorrect naming of variables to improve code readability and maintainability.

OS-WHS-SUG-00 | Duplicate Guardian Check

Description

Wormhole implements `guardian_set` by storing elements in an array. This implementation does not guarantee the uniqueness of `guardians` within the set. It allows room for governance mistakes leading to adding the same `guardian` in the set more than once. Such mistakes would grant specific `guardians` greater voting power, potentially leading to decentralization concerns.

Remediation

Check against the updated `guardian_set` to ensure no duplicate entries exist.

sui/wormhole/sources/resources/guardian_set.move DIFF

```
  public fun new(index: u32, guardians: vector<Guardian>): GuardianSet {
+     // Ensure that there are no duplicate guardians.
+     let (i, n) = (0, vector::length(&guardians));
+     while (i < n - 1) {
+         let left = guardian::pubkey(vector::borrow(&guardians, i));
+         let j = i + 1;
+         while (j < n) {
+             let right = guardian::pubkey(vector::borrow(&guardians,
    ↪  j));
+             assert!(left != right, E_DUPLICATE_GUARDIAN);
+             j = j + 1;
+         };
+         i = i + 1;
+     };
      GuardianSet { index, guardians, expiration_timestamp_ms: 0 }
  }
```

Patch

Resolved in 760db3c.

OS-WHS-SUG-01 | Minimize External Attack Surface

Description

Sui supports `friend` functions where only a set of whitelisted friend modules are allowed to call them.

We generally suggest marking all object-modifying functions that are not intended to be used by external users as `friend` only to minimize the potential attack surface.

For instance, the function below could benefit from stricter access control.

sui/wormhole/sources/resources/set.move RUST

```
public fun add<T: copy + drop + store>(self: &mut Set<T>, key: T) {
    assert!(!contains(self, key), E_KEY_ALREADY_EXISTS);
    table::add(&mut self.items, key, Empty {})
}
```

Remediation

Ensure functions that should only be used internally are `friend` only.

sui/wormhole/sources/resources/set.move DIFF

```
- public fun add<T: copy + drop + store>(self: &mut Set<T>, key: T) {
+ public(friend) fun add<T: copy + drop + store>(self: &mut Set<T>, key:
    ↪  T) {
      assert!(!contains(self, key), E_KEY_ALREADY_EXISTS);
      table::add(&mut self.items, key, Empty {})
  }
```

OS-WHS-SUG-02 | Fix Misnamed Variables

Description

The code currently includes misnamed variables that may lead to confusion in code analysis and future maintenance.

Remediation

Assign proper names to variables.

sui/token_bridge/sources/governance /upgrade_contract.move DIFF

```
  use token_bridge::state::{Self, State};
  ...
  fun handle_upgrade_contract(
-     wormhole_state: &mut State,
+     token_bridge_state: &mut State,
      msg: GovernanceMessage
  ): UpgradeTicket {
      // Verify that this governance message is to update the Wormhole
     ↪  fee.
      let governance_payload =
          governance_message::take_local_action(
              msg,
              state::governance_module(),
              ACTION_UPGRADE_CONTRACT
          );

      // Deserialize the payload as amount to change the Wormhole fee.
      let UpgradeContract { digest } = deserialize(governance_payload);

      state::authorize_upgrade(wormhole_state, digest)
  }
```

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	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 implementation of the program 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 sum, 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 get a comprehensive understanding of 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.