OtterSec

Lombard Finance

Security Assessment

December 6th, 2024 — Prepared by OtterSec

Michał Bochnak embe221ed@osec.io

Sangsoo Kang sangsoo@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Lombard Finance engaged OtterSec to assess the `sui-contracts` program. This assessment was conducted between December 2nd and December 5th, 2024. 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 critical vulnerability, where the minting function incorrectly resets the remaining mint limit during a new epoch, as it assigns the limit value directly instead of referencing (OS-LBF-ADV-00), and another issue concerning upgrade authorization function, which utilizes a hardcoded delay of 24 hours instead of the configurable delay, limiting its flexibility and disregarding custom delay settings (OS-LBF-ADV-01).

We also made recommendations for modifying the codebase to improve functionality and prevent unexpected outcomes (**??**).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/lombard-finance/sui-contracts. This audit was performed against commit 45400c0.

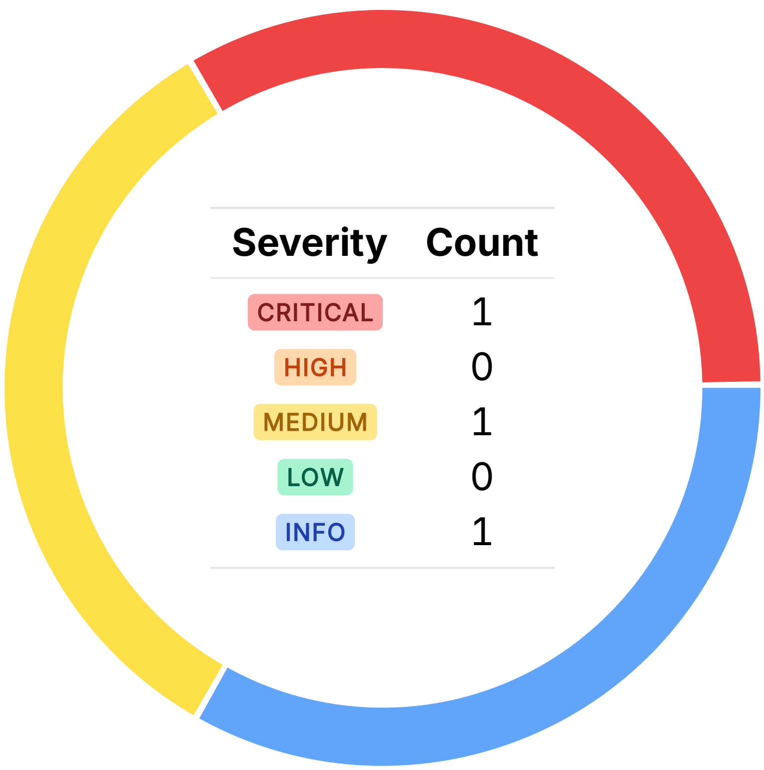
A brief description of the programs is as follows:

Name	Description
sui-contracts	The Sui contracts of the Lombard Finance Protocol bridge Bitcoin into DeFi through LBTC, a regulated, yield-bearing token backed 1:1 by BTC.

03 — 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	1
HIGH	0
MEDIUM	1
LOW	0
INFO	1

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-LBF-ADV-00	CRITICAL	RESOLVED ✓	The minting function incorrectly resets the remaining mint limit (`left`) during a new epoch, as it assigns the `limit` value directly instead of referencing it with `*limit`.
OS-LBF-ADV-01	MEDIUM	RESOLVED ✓	`authorize_upgrade` utilizes a hardcoded delay (`MS_24_HOURS`) instead of the configurable `timelock.delay_ms`, limiting its flexibility and disregarding custom delay settings.

Improper Mint Limit Reset CRITICAL OS-LBF-ADV-00

Description

In `treasury::mint_and_transfer`, the line `left = limit;` modifies the local variable `left`. However, `get_cap_mut(treasury, ctx.sender())` returns a mutable reference to the `MinterCap` object associated with the sender. This implies `left` is a mutable reference, which refers to the actual value in the `MinterCap` structure. Thus, currently, the function is only re-assigning the `left` variable with a reference to the `limit` field of the structure rather than updating the `left` field. So, to properly update the value of `left` within the `MinterCap` structure, it needs to be de-referenced by utilizing `*left`.

>_ lbtc/sources/treasury.move MOVE

```move
public fun mint_and_transfer<T>(
   [...]
) {
    [...]
    // Get the MinterCap and check the limit; if a new epoch - reset it
    let MinterCap { limit, epoch, mut left } = get_cap_mut(treasury, ctx.sender());
    // Reset the limit if this is a new epoch
    if (ctx.epoch() > *epoch) {
        left = limit;
        *epoch = ctx.epoch();
    };

    // Check that the amount is within the mint limit; update the limit
    assert!(amount <= *left, EMintLimitExceeded);
    *left = *left - amount;
    [...]
}
```

Remediation

Set `*left = *limit` instead of `left = limit` to correctly update the available minting allowance.

Patch

Fixed in ecf55e3.

Lack of Configurable Delay Setting in Timelock MEDIUM OS-LBF-ADV-01

Description

`timelock_upgrade::authorize_upgrade` utilizes a fixed delay of `MS_24_HOURS` (24 hours) to enforce the time restriction on upgrades, rather than referencing the configurable delay stored in `timelock.delay_ms`. This introduces inconsistency and defeats the purpose of having a customizable delay feature.

>_ *timelock_policy/sources/timelock_upgrade.move* MOVE

```move
public fun authorize_upgrade(
    timelock: &mut TimelockCap,
    policy: u8,
    digest: vector<u8>,
    ctx: &mut TxContext,
): UpgradeTicket {
    let epoch_start_time_ms = ctx.epoch_timestamp_ms();
    assert!(
        timelock.last_authorized_time == 0 || epoch_start_time_ms >=
            ↪ timelock.last_authorized_time + MS_24_HOURS,
        ENotEnoughTimeElapsed,
    );

    timelock.last_authorized_time = epoch_start_time_ms;
    timelock.upgrade_cap.authorize(policy, digest)
}
```

Remediation

Utilize `timelock.delay_ms` instead of `MS_24_HOURS` for better customizability.

Patch

Fixed in d2e3a5d.

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-LBF-SUG-00	There are several instances where proper validation is not done, resulting in potential security issues.

Missing Validation Logic

OS-LBF-SUG-00

Description

1. Include the assertion: `assert!(delay_ms == MS_24_HOURS || delay_ms == MS_48_HOURS)` with the `EInvalidDelayValue` error, in `new_timelock` to validate that the provided delay value is one of the allowed options (24 hours or 48 hours).

   >_ *timelock_policy/sources/timelock_upgrade.move* MOVE

   ```move
   /// Creates a new TimelockCap with the specified delay.
   public fun new_timelock(
       upgrade_cap: UpgradeCap,
       delay_ms: u64,
       ctx: &mut TxContext,
   ): TimelockCap {
       TimelockCap {
           id: object::new(ctx),
           upgrade_cap,
           last_authorized_time: 0,
           delay_ms,
       }
   }
   ```

2. In `treasury`, verify the length of the `pks` vector elements to ensure that each element of `pks` is a valid public key before it is utilized in the multisig address validation.
3. `mint_and_transfer` and `burn` in `treasury` do not validate whether the amount is greater than zero, which impacts the correctness of these operations. Allowing zero-value minting or burning is unnecessary, wastes computational resources, and adds noise to event logs. Add validation to ensure if `coin.value() > 0` in `burn` and `amount > 0` in `mint_and_transfer`.

Remediation

Incorporate the above-mentioned validations into the codebase.

Patch

1. Issue #1 fixed in d2e3a5d.
2. Issue #2 fixed in 00d1dfc.
3. Issue #3 fixed in 0f00717.

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.